

# Contents— Overview

---

## SECTION I BEGINNINGS

Section I covers C and assembly, for the 8051 family. It will help to have some previous experience programming. To keep the examples simple, this section uses only parallel ports.

Chapter 1 introduces the 8051 microcontroller and sets it in the context of other micros.

Chapter 2 describes computer architecture in general and that of the 8051 family specifically. It describes kinds of memory and explains how information travels over a bus. The way the central processing unit (CPU) does math and logical operations is described here.

Chapter 3 goes through *all* the machine instructions of the 8051 family. While brief examples are given, the following chapters are the place to learn programming.

Chapter 4 introduces assembly language and C. The different kinds of variables and the different types of memory space in the 8051 are described. The chapter covers the logical and arithmetic operations that are important to embedded applications. The precedence of operators is shown in a table.

Chapter 5 covers the branching and looping constructs, which are essential to any structured programming approach. The idea of structured programming is explained as well as the difference between a loop test at the start or end of a loop.

Chapter 6 gets into arrays, pointers, and based variables, which are fundamental to functions (and usually come at the end of a programming course). This chapter also goes into structures.

## SECTION II FUNCTIONS, MODULES, AND DEVELOPMENT

Chapter 7 covers *functions* and *subroutines*—the pieces that contribute to modular, understandable programs. This chapter also goes into passing values to and from routines.

Chapter 8 goes into scope of variables, using multiple files in developing software and the mixing of languages. The modular approach is no longer just a technique for switching to assembly when a high-level language becomes too slow—it is the key to well-organized, easily maintained programs.

Chapter 9 introduces the “integrated development environment” and software tools from Keil that come with the book. It shows the development process and then how to set up the environment so you can type in programs and have them compiled (or assembled) to the final form. There is material on the two monitor programs that come with the available development boards as well.

## SECTION III MULTITASKING

If you are going to write efficient embedded applications code, you must venture forth on new way of thinking in which your controller does multiple things “at the same time.” This should be the heart of any real-time control projects.

Chapter 10 introduces the *task* and related terms of multitasking.

Chapter 11 introduces the timer and interrupt hardware of the 8051 family that are the key to real-time interrupts and most multitasking.

Chapter 12 develops a form of multitasking system—the scheduler. It shows how the real-time interrupt makes programming of traffic lights and other cyclic controllers quite straightforward.

Chapter 13 categorizes and describes real-time operating systems as communication between tasks on the same controller. In addition to the simplest methods using flags and shared variables, this chapter describes signaling, message passing and resource management in the various multitasking operating systems.

Chapter 14 goes through a specific example showing the planning, hardware choices, and the software development. It shows how timers and interrupts allow a sort of multitasking.

## **SECTION IV APPENDICES**

A1 gives both numeric- and alphabetic-order lists of assembly language instructions for the 8051. This is useful if you are hand-disassembling machine code.

A2 covers development under DOS as well as describing some of the batch files for developing software for use with a monitor.

A3 gives language-switching hints—8085 to 8051 assembly as well as ANSI-standard C to C with 8051 extensions.

A4 describes bank switching as well as covering the design and use of the two featured development boards.

A5 lists all the known vendors of 8051-related products with current addresses and phone numbers.

# Contents

---

**PREFACE**   xix

## **I BEGINNINGS**   1

### **1 Introduction**   3

- What is an “8051”?   3
- Microcontrollers   4
- Programming Embedded Controllers in C   5
  - Programming Languages*   5
- Efficiency   5
- Review and Beyond   6

### **2 Computer Basics**   7

- Overview   7
- Dissecting C: Back to Basics   8
  - C or Assembly?*   11
- Computer Architecture   11
  - Binary Numbers*   13
  - Address Decoding*   14
    - Single-address Decoder*   14
    - Multidevice Address Decoder*   15
  - PLDs*   17



<i>Memory</i>	17
<i>ROM and RAM</i>	18
<i>8051 Code Storage</i>	20
<i>8051 Internal RAM</i>	21
<i>8051 Off-chip Memory</i>	21
<i>Input/Output (I/O)</i>	22
<i>Moving Data—Busses</i>	24
<i>8051 Instruction Execution Sequences</i>	26
Timing and Signal Details	28
<i>Control Bus Signals</i>	28
<i>ALE</i>	28
<i>PSEN</i>	29
<i>RD and WR</i>	30
<i>The Oscillator</i>	31
<i>Clocks and Timing</i>	31
<i>Clock Cycles, Machine Cycles, and Instruction Timing</i>	32
<i>Register Banks</i>	33
<i>Special Function Registers</i>	33
<i>Arithmetic Logic Unit (ALU)</i>	34
Stand-alone Microcontrollers	38
Memory Expansion for the 8051	39
<i>Off-chip Code</i>	39
<i>Off-chip Code, Data, and Ports</i>	41
<i>Off-chip Code, Data, Ports, and A-D</i>	45
<i>Port-driven Peripherals (LCD)</i>	45

### **3 Machine Instructions 49**

Data Moving Instructions	59
<i>Accumulator/Register</i>	62
<i>Accumulator/Direct</i>	62
<i>Accumulator/Data</i>	62
<i>Register/Data</i>	62
<i>Accumulator/Indirect</i>	62
<i>Register/Direct</i>	62
<i>Direct/Direct</i>	62
<i>Direct/Indirect</i>	62
<i>Direct/Data</i>	63
<i>Indirect/Data</i>	63
<i>Data Pointer/Data</i>	63
<i>MOV</i>	63

<i>MOVX</i>	64
<i>XCH</i>	64
<i>Accumulator/Register</i>	64
<i>Accumulator/Direct</i>	64
<i>Accumulator/Indirect</i>	64
Stack Instructions	65
<i>PUSH</i>	65
<i>POP</i>	66
Branching Instructions	66
<i>Unconditional JMP</i>	66
<i>Conditional JMP</i>	67
<i>Comparisons</i>	67
<i>CJNE</i>	67
<i>DJNZ</i>	68
<i>Calls</i>	68
<i>ACALL</i>	69
<i>LCALL</i>	69
<i>RET</i>	70
<i>RETI</i>	70
<i>No Operation—NOP</i>	70
Logic instructions	70
<i>ANL</i>	70
<i>ORL</i>	71
<i>XRL</i>	71
<i>CPL</i>	71
<i>CLR</i>	71
<i>Rotating</i>	71
<i>RR, RL, RRC, RLC</i>	72
Boolean (Bit) Instructions	72
<i>Flags and Their Uses</i>	72
<i>CLR (Bit)</i>	74
<i>SETB</i>	74
<i>CPL (Bit)</i>	74
<i>ANL (Bit)</i>	74
<i>ORL (Bit)</i>	74
<i>MOV (Bit)</i>	75
Math Instructions	75
<i>Addition</i>	75
<i>Subtraction</i>	76
<i>Other Math</i>	76
<i>Decimal Instructions</i>	77

Wrapping Up the Instruction Set	78
<i>Writing Assembly Language</i>	78
Exercises in Assembly language	79
<i>Accumulator/Register</i>	84
<i>Accumulator/Direct</i>	84
<i>Accumulator/Data</i>	84
<i>Register/Data</i>	84
<i>Accumulator/Indirect</i>	84
<i>Register/Direct</i>	84
<i>Direct/Direct</i>	84
<i>Direct/Indirect</i>	84
<b>4 Two Languages</b>	<b>84</b>
Why these two?	84
Variables	86
Shorthand: #define or EQU	88
Memory Spaces	89
Ports	92
Example: Switches to Lights	92
Bitwise Logical Operators	98
Rotate and Shift	100
Assignment Operators	101
Identifying Bit Changes	102
Arithmetic Operators	106
Logical Operators	113
Precedence	114
Review and Beyond	115
<b>5 Looping and Branching</b>	<b>117</b>
Decisions	117
Flowcharts	117
Structured Language	118
Branching Constructs	120
<i>If/else</i>	120
<i>Conditional Operator</i>	122
<i>Switch</i>	122
<i>Breaking Out: The Goto</i>	124
<i>Looping Constructs</i>	124
<i>While Loop</i>	124
<i>Iterative Loop</i>	126

Example: Time delay 128

Review and beyond 129

## **6 Arrays and Pointers 130**

Arrays 130

Lookup tables 133

Structures 134

New Data Types: typedef 136

Array of Structures 136

Arrays within Structures 137

Choosing Memory Spaces for Variables 138

Pointers 139

Universal Pointers 141

Array Pointers 142

Arrays of Array Pointers 142

Structure Pointers 144

Unions 145

Review and Beyond 146

## **II FUNCTIONS, MODULES, AND DEVELOPMENT 149**

## **7 Functions 151**

Subroutines, Procedures, and Functions 151

Functions Ease Understanding 154

*Drivers* 156

Nested Functions 156

Passing parameters 157

Example: Write to LCD Module 158

*Writing to LCD From Ports* 158

*Writing to LCD as Memory* 158

Returning Values 163

Example: Scan a Keyboard 164

Example: Read an A-D Converter 167

In-line Code Alternatives 170

Scope of Variables and Functions 170

Review and Beyond 173

## **8 Modular Programming 174**

Why Modular Programming? 174

Terms and Names	174
Sharing variables	178
<i>C Variable Scope Conventions</i>	178
<i>C Function Scope Conventions</i>	179
<i>Assembly Scope Conventions</i>	180
Single-Language Modules	180
<i>Modular ASM51 Example: Stepper Driver</i>	180
Mixing Languages	190
<i>Parameter-Passing Conventions</i>	191
<i>Compatibility by Test</i>	192
<i>Mixed Language Example: Stepper Driver</i>	198
<i>Mixed Language Example: Muth</i>	203
Libraries	208
<i>The Standard C Libraries</i>	209
<i>Your Own Library</i>	209
<i>Functions for a Library</i>	210
<i>Keep Variables Private</i>	210
<i>Function Prototypes in a Header File</i>	210
<i>Making the Library</i>	211
Shortcuts	214
<i>Code Efficiency</i>	214
<i>Headers for Register and I/O Definitions</i>	216
<i>Off-chip Variables</i>	217
<i>Overlaying</i>	217
<i>More about Linking</i>	217
Review and Beyond	218
<b>9 Development and Debugging</b>	<b>219</b>
The Overall Development Sequence	220
Developing Software with $\mu$ Vision	221
<i>Defining a Project</i>	224
Installing $\mu$ Vision	228
<i>Setting Parameters</i>	229
Development Tools	240
<i>Simulators: DS51</i>	241
<i>Monitors</i>	246
<i>EET MONITOR PROGRAM V1.8 (EET Monitor)</i>	
<i>for 8051 Family Microcontrollers</i>	249
<i>Commands to Display and Change Memory, Registers</i>	
<i>and I/O</i>	251

<i>Running and Debugging Programs</i>	252
<i>NDTES</i>	254
<i>EET Monitor Subroutines</i>	254
<i>Use of C Programs with the EET Monitor</i>	225
<i>Emulators</i>	260
<i>PROM Programmers</i>	261
Debugging Strategies	262
<i>Test Hardware First</i>	263
<i>Test the Processor</i>	263
<i>Test the Ports</i>	263
<i>Growing Software</i>	264
<i>Start Small</i>	264
<i>Avoid "Burn and Try" (Develop First On a Higher-Powered Relative)</i>	265
<i>Use Breakpoints</i>	265
<i>Use I/O Pins as Scope Trigger Points</i>	266
Review and Beyond	267

## IV MULTITASKING 269

### 10 Concepts and Terms 271

Beyond Single-Program Thinking	271
What Is "Real-time"? 271	
Job	273
Task	274
<i>Priority</i>	274
Preemption	274
Multitasking	275
<i>Cooperative Multitasking—Round Robin</i>	275
<i>Time-slice Multitasking</i>	276
<i>Multitasking with a Scheduler</i>	276
<i>Priority-based, Preemptive Multitasking</i>	277
<i>Events</i>	277
Real-time Hardware Requirements	277
<i>Timer</i>	277
<i>Interrupt</i>	278
<i>Real-time Clock</i>	278
Programming Habits For ALL Multitasking	278
<i>Never Wait</i>	278

<i>Set Flags</i>	279
<i>Be Sure Variables are Global</i>	279
<i>Avoid Software Loops</i>	279
Review and Beyond	279

## 11 Timers, Interrupts, and Serial Ports 281

Counter/timers	281
<i>Internal Timer Details</i>	283
<i>Example: A 1usec Timer</i>	285
<i>Other Modes</i>	286
<i>Timer2</i>	286
Interrupts	288
<i>How an Interrupt Works</i>	288
<i>Masking and Interrupt Enables</i>	290
<i>External Interrupt Hardware</i>	291
<i>Expanding Interrupts</i>	291
<i>Interrupt Priorities and Latency</i>	293
<i>Context Switching</i>	295
<i>Real-time Clock</i>	296
Fast Events and High Frequencies	298
<i>Infrequent Events and Low Frequencies</i>	299
<i>In-between Frequencies</i>	301
<i>Broad-Range Frequencies</i>	301
Serial Ports: The 8051's UART	302
<i>Example: Serial Buffering</i>	304
<i>Shift Register Mode</i>	310
<i>Ninth-bit Mode</i>	310
Review and Beyond	311

## 12 Build Your Own Scheduler 312

<i>Example: Traffic Light (Basic Cycle)</i>	312
<i>Example: Traffic Light (With Walk Buttons)</i>	316
More Elaborate Communication	318
Break-out Functions	318
Communication: Shared Variables	320
Nothing to Do	320
<i>Example: Solenoid Cyclere</i>	321
<i>Example: Pulse Generator</i>	322
<i>Example: Envelope Detector</i>	324

Delayed Tick Due to Overload	326
Catching All Ticks	327
Review and beyond	327

### 13 Real-time Operating Systems 328

Why an Operating System for Multitasking?	329
How do Operating Systems Really Work?	329
<i>Context Switching</i>	330
<i>Setting Priority</i>	332
<i>Other Interrupts and Interrupt Handlers</i>	333
Resources, Regions, Pools, and Lists	334
Communication and synchronization	334
<i>Shared Variables for Communication</i>	335
<i>Drawbacks of Shared Variables</i>	335
<i>Counting Semaphore</i>	336
<i>Messages with Operating System</i>	337
Commercial operating systems	337
<i>DCX51</i>	337
<i>RTX51/RTXtiny</i>	339
<i>Two Basic Groups of RTOS</i>	339
<i>USX</i>	339
<i>CMX</i>	340
<i>Byte-BOS</i>	341
<i>Observations on USX, CMX, and Byte-BOS</i>	341
<i>Benefits of RTOS</i>	342
<i>Costs of RTOS</i>	343
Review and beyond	343

### 14 Putting It All Together: An Example 344

Circulating Hot Water Pump Controller	345
<i>The Problem: Delayed Hot Water</i>	345
<i>Background: Hot Water Systems</i>	345
The Solution: Automatic Pump Controller	346
<i>Specifications</i>	346
<i>User's Instructions</i>	348
Principles of Operation	349
<i>Flow Sensor</i>	349
<i>Temperature Sensor</i>	349
<i>Sensor Interface</i>	350
<i>Microcontroller</i>	351



<i>Multitasking Features</i>	354
<i>Pump Drive</i>	358
<i>Power Supply</i>	358
<i>Status Indicator</i>	359
<i>Mode Switch</i>	359

## **IV APPENDICES 361**

### **A1 363**

Numeric Order	363
Instructions Sorted Alphabetically	370

### **A2 Development with DOS 377**

Source Code Entry	377
Locating Code and Variables	377
Compiling	378
Assembling	379
Linking	380
Hex conversion	380
Batch Files	381
Space-Reserving Files	382
Libraries	384

### **A3 Language-Switching Hints 387**

Other Assembly to 8051 Assembly	387
PL/M to C	388
Standard (ANSI) C to C51	389

### **A4 Boards 390**

Bank-switching Code	391
PU552 Microcontroller Board	393
Hook-up/Connections	396
Operation	399
MCB520 Evaluation Board	400
<i>Switch Settings for the MCB520 Board</i>	401
<i>Writing Programs for the MCB520 Board</i>	402
<i>Internal Port Availability</i>	403
<i>Schematic</i>	403
<i>Other Commercial Boards</i>	405

## **Index 427**

# Preface

---



If you are developing microcontroller-based electronics, this book is for you. The objectives of this book are to start you programming embedded applications and to help you develop the mindset of modules and multitasking. Programming examples in this book mostly use *internal* peripheral devices found within the 8051 family of microcontrollers as well as a few common *external* devices. In a companion book, *C and the 8051: Building Efficient Applications*,<sup>1</sup> I apply microcontrollers to a large variety of peripheral hardware and use

more advanced multitasking. The companion book assumes you can program and concentrates on application details. Throughout both books I emphasize efficiency and good project development procedures.

Despite the contempt of technophiles because they run slowly, access much less memory, and seldom have elaborate math capability built in, embedded microcontrollers continue to be *much less expensive than the faster, bigger processors!* Instead of costing the hundreds of dollars of a Pentium-class processor, an 87C750, a complete computer in a 24-pin socket, costs perhaps \$4 plus the cost of a crystal.<sup>2</sup> I do not want to detract from the im-

<sup>1</sup>Despite my initial ambitions, the companion book is still very rough. Prentice Hall agrees, the goal is to have the book released before the end of 1998.

<sup>2</sup>For an example see the final chapter (Chapter 14) where an embedded controller and three other chips make up an entire control system.

portance of the high-power areas; rather, I illustrate the ideas and techniques that set the *embedded* controller field apart from the large-computer-system emphasis.

## EMBEDDED MICROCONTROLLERS

What is an embedded microcontroller? Like a sliver embedded in your finger, an embedded microcontroller is not visible from the outside, but it too has a significant effect. Embedded devices are just the opposite of a PC<sup>1</sup>—no one programs them after they arrive. In a microwave oven, for example, the embedded controller comes programmed knowing which output turns on the oven, which input comes from the door interlock, and so on. Day after day the controller runs the same program—the chip in the microwave will never control automobile ignition or play a video game because it is pre-programmed. You could program the same *type* of chip for those other jobs, but the embedded one in the microwave is not a general-purpose computer.

An embedded *real-time* controller probably runs only one program for its entire life. It must be always ready to handle any number of inputs and outputs at the “same” time. Unlike a PC, the hardware is more likely to be switches and solenoids than disk drives and keyboards.

The software examples in this book relate to embedded controllers rather than to general-purpose computers for data manipulation or graphics. There is still a corner of the programming field where the efficient approaches developed with small, dedicated microcontrollers are best. You do not need an elaborate set of print functions in a system where the only output is a few LEDs and a motor!

---

<sup>1</sup>A PC or personal computer is not conceptually different from an embedded controller. While it is now much more complex, it has the same architecture as a microcontroller, runs programs in the same way (but faster), and has input and output. The difference is that you use a PC to do different things at different times. You would have a problem if your car engine controller instead chose to plot graphics for a while. Typical inputs and outputs are different too. With a PC you usually supply input from a keyboard or mouse and output goes to a display or printer. You also involve large amounts of storage—floppy and hard disks, backup tapes, and CD-ROMs. Embedded controllers *can* do all of these things in demanding applications and might even be the same processors, but the fundamental difference is the dedicated nature of the system. As evidence of their equivalence, there is occasional talk of putting telephone switching systems (embedded processors for sure) to use in the off hours doing business computing.

## FOR MORE THAN THE 8051 FAMILY?

This book concentrates on the 8051 family. Trying to cover all the different microcontrollers (68HC11, PIC, 8096, 68HC16, etc.) would make the examples very confusing since the assembly languages and the hardware specifics differ. However, concentrating on only one processor limits the number of people that can use the book. By discussing the *principles* involved with flowcharts and pseudo-code before the specifics of the examples, I keep the examples as broad as possible. If you can find an appropriate C compiler, you can easily transfer the C examples, despite the 8051 language extensions, to other microcontroller families.

Since most microcontrollers have similar internal timer and port features, the examples using internal features can often be adapted. While different commands set up the timers, and interrupts may behave somewhat differently, the basic principles remain the same.

## IS THIS BOOK FOR YOU?

### Students—Classes

You can use this book in both beginning and advanced microcontroller classes. The first section introduces you to the basics of computer hardware and programming. Chapter 3 covers assembly language and Chapters 4 to 6 cover basic programming constructs. It helps to have prior programming experience (no matter what the language) but you *can* use this book to first learn programming in either assembly language or C. An assembler and linker as well as a full-featured but code-size limited version of a C compiler comes with this book. The circuit boards used in the examples are easy to obtain for lab, classroom, or home use.

### Students—Design Projects

In a design course, this book can get you started putting a micro to intelligent use in your project. Elegant design is far different from relying on raw computing power and speed. The software and, the available boards supply all you need to get going with embedded controller development.<sup>4</sup> If

---

<sup>4</sup>If you already know programming, the companion book, *C and the 8051: Building Efficient Applications*, is probably a more useful reference. It is rich in application examples, circuit ideas and schematics. If you are going to skip to the companion book, you should be familiar with either C or assembly programming for the 8051 family.

you are going to build or adapt the electronic applications, you should have some knowledge of circuit breadboarding and assembly techniques. Knowledge of digital and analog electronics would help too!

### **Engineers/Developers**

Engineers or system designers producing small embedded applications should find this book and the companion immediately applicable. If you are the sort of person who prefers to learn through the practical hardware-oriented examples and who may be looking for new ways to replace existing hardware with a micro, this is the book to have.<sup>5</sup>

If you are already developing 8051 embedded systems, you may still benefit from the sections on modular programming and multitasking. Also knowing the Keil/Franklin tools (Chapter 9) might be useful—things have changed significantly from the DOS-based tools of a few years ago.

### **Self-Taught Home Experimenter**

If you are an "experimenter/hobbyist," you want to see interesting applications running right away—not read about fine points of design. Despite some reviews of the previous edition, this book is not really just a "designer's book." Unlike most engineering fields, advanced math and college training are not very important with microcontroller applications. You might want to start by getting or wiring up the example board and trying the programming examples. Since the software comes with the book, once you have the board, a serial cable, and a PC, you have all you need to get started.<sup>6</sup> Unlike most magazine-article projects, with the software and information from this book, you will not have to buy a pre-programmed EPROM for your projects—you will be writing your own!

No matter which group you put yourself in, you need a book of ideas along with enough specifics to get going without a lot of wasted effort. That is my goal.

---

<sup>5</sup>Especially in the companion book I describe new approaches you can store away for future projects.

<sup>6</sup>As a home experimenter you are probably already experienced with electronic construction, so you can make good use of the companion book full of applications and software to go with them.

## THE INCLUDED SOFTWARE

The disk included with this book has Keil's assembler, C compiler, and linker. The C compiler is full-featured but the supplied linker only allows 2K of code. Companies such as Keil still hope to make their living selling the software to industrial developers. You can compile larger programs to check code size, but for the final code production, you will have to make an additional purchase.<sup>7</sup> If you want copies of the software examples from the book, Keil has indicated a willingness to make them available for downloading over the Internet.

## THE AVAILABLE HARDWARE

Many of the examples in the book fit the two microcontroller boards described in Appendix A4. You may purchase the two boards only in populated/tested form. I was hoping to include a blank board with every book, but economic and marketing realities reduced that to schematics.<sup>8</sup>

## WHAT THIS BOOK IS NOT

This book is not a general book on computing. A beginner's tutorial on programming is different from this book covering computer hardware, machine instructions, and the C language.

There may be books that do a more thorough job with the details of assembly language programming. While I have included numerous assembly

---

<sup>7</sup>A slightly bigger version is available from Keil Software for a few hundred dollars. The compiler is among the best on the market for the 8051 family, but the full professional version (as well as all the competing professional products) sells for between \$1,000 and \$2,000. There are some less expensive compilers, but they lack features or behave much less efficiently with the (rather unusual) architecture of the 8051. You may also contact the author by telephone at (765)494-7724 or by e-mail at [tweschultz@tech.purdue.edu](mailto:tweschultz@tech.purdue.edu)

<sup>8</sup>You have a schematic in the appendix if you feel qualified to wire your own board, but troubleshooting (if something doesn't work right away) requires equipment you might not have. The original plan was to supply the copper pattern and offer bare boards so you could build replicas of the pre-built board. Keil has indicated a willingness to offer bundled packages of multiple (assembled) boards and software tools for school use and I expect Rostek would do the same.

## SECTION I

# Beginnings

---

Start here if you need a foundation in computer architecture and machine instructions. This section starts with the basic 8051 hardware and the machine instructions. Knowing the *instructions* is not the same as knowing how to program any more than knowing the *alphabet* means you can read a language, but it is a good place to start.

Chapter 1 introduces the area of embedded controllers and their programming.

Chapter 2 works backward from a C program to the computer architecture and digital logic showing how: 1) machine (assembly) instructions arise from a C program you might write; 2) machine instructions arise out of computer architecture; 3) computer architecture comes from arrangements of digital gates.

If you have had no exposure to computer hardware, I suggest you skim Chapters 2 and 3 before going on to Chapter 4. Although Chapter 3 discusses assembly language instructions, I do not do much with them there—not much putting them together to make programs. Understanding Chapters 2 and 3 is not absolutely necessary to write C programs for the 8051. However, while people speak of writing “platform-independent” (ANSI standard) C, the 8051 hardware is too *nonstandard* to get efficient programs that way.

Chapters 4 through 7 cover the two languages featured—C and assembly language. After the most basic constructs, they describe looping and branching (Chapter 5) and arrays (Chapter 6). Here is where I go over program constructs—the things you *do* regardless of the language involved. I show all the examples in C and usually in assembly as well.

# 1

## Introduction

---

### WHAT IS AN “8051”?

The 8051 number applies generically to a family of microcontrollers that was the successor to the 8048—the first single-chip microcontroller. The original version of the 8051 developed by Intel has been on the market for over fifteen years.<sup>1</sup> The architecture and instruction set of the 8051 live on in what is now a large family of derivative microcontrollers having more code space, more timers, more interrupts, and more peripherals. Dallas Semiconductor has developed versions that take the same instructions but run in fewer clock cycles and have additional architectural features. Both Intel and Phillips have developed 16-bit “relatives” that are direct migration paths in that they are either code or assembly-language compatible.<sup>2</sup> Going the other direction, Philips has developed derivatives that are smaller and less expensive. The wide base of development tools and experience with the 8051 family has made it the most common choice for middle-of-the-road projects needing embedded control.

---

<sup>1</sup>Because of growing expectations, many commercial applications no longer fit in the original 4K on-chip code space. Some applications no longer even fit in the 64K total addressable code space and have gone to switching between banks of memory, discussed in Appendix A4.

<sup>2</sup>Intel’s 80C251 family and Philips’ 8051XA.



## MICROCONTROLLERS

The distinction between a *microcontroller* and a general purpose computer or *microprocessor* is not a sharp one. They all have a processor as part of the system to run the program—looking at each instruction to decide what to do, doing it, and going to the next instruction. To be useful, the processor must have memory and input/output (I/O) capability as well. If this is mostly on one chip, it is called a microcontroller. If, along with the microprocessor chip,<sup>3</sup> you need several other chips for bus interface, memory, and I/O, the entire system is a *microcomputer*.<sup>4</sup>

Most programs can run on *any* of the 8051 family. While the book's title mentions the 8051, the examples, particularly in the companion book, show many other members of the continually growing family of related devices. The basic "core" remains the same—the differences relate to the built-in peripherals. This book works almost totally with the basic 8051 features.<sup>5</sup> Most examples are applicable to the *entire* family because they rely on the 8051 core.<sup>6</sup>

---

<sup>3</sup>A *chip* is another term for an integrated circuit (IC). The name comes from the fact that a thin, round slice of silicon (a wafer) is optically, chemically, or electronically treated to get a lot of repeated circuits. Then the wafer is cut apart into lots of individual, rectangular circuits somewhat like the chips of wood that come from chopping down a tree. These chips are mounted in plastic or ceramic chip holders with wires leading out to the legs for the conventional *dual inline (DIP) packages*, or in much smaller configurations for the newer *surface-mount* devices.

<sup>4</sup>A *minicomputer* was a higher performance computer made up of even more, faster chips, but the term has died as microcomputers have increased in performance.

<sup>5</sup>In the companion book some applications, where small size and low cost are important, use the 87C750. Other applications there use additional features of the 517, 552, or 558. Still other applications illustrate the memory features of the DS5000 series with on-chip "off-chip" code and data RAM. Where the attention is on analog interfacing, some examples use the on-chip AD and DA found in some of the devices. I supplement these with examples using traditional add-on devices. The part on timers now includes examples of more advanced timer modes available. One section in the companion book deals specifically with the 1<sup>2</sup>C Bus devices.

<sup>6</sup>Other microcontroller families: Today, most embedded applications use 8-bit controllers—they give a high level of flexibility without the high cost. The leaders in this area are probably the 8051 family and Motorola's 68HC11. Some of these processor cores are so common in low-to-medium performance applications that they come in an ASIC (application specific integrated circuit) design library much the same way as you could add a flip-flop or counter design to a custom-integrated circuit. The 8051 really has become a standard in the microcontroller world.

If you need *high-end* (high-speed) capability, there seem to be new 16- and 32-bit controllers coming out yearly with no clear "winner" as of yet. Interestingly enough, Intel has recently developed the 16-bit 80C251 microcontroller that retains code compatibility with the

## PROGRAMMING EMBEDDED CONTROLLERS IN C

### Programming Languages

What language should you use? Three languages are common with the 8051 family—C, BASIC, and assembly. In addition, for many years there has been a little use of Forth and a significant use of PL/M for programming. Over the last five years, at least, the trend has been to switch to C. Most developers now rely on the C language. Along with the benefits of high-level languages like C, the elaborate software tools of today represent a dramatic shift from the difficult, detail-oriented programming techniques necessary for the first microprocessors. Most books teaching C deal with the ANSI standard language running data processing (big computer or PC) examples, which are inappropriate to the 8051. In this book all the programming examples use C, and the simpler ones also use assembly language. Appendix A3 discusses conversion from PL/M to C.

While knowing assembly language may not be your goal, seeing and understanding a little of it will help you understand the limitations of the 8051. For example, knowing how the 8051 assembly language instructions access memory spaces makes the advantage of on-chip RAM for variables quite obvious.<sup>7</sup> All the examples have compiled or assembled properly and have run with a simulator or with actual hardware, so you can be reasonably confident that they work.

### EFFICIENCY

Presumably you want to produce the most user-friendly, reliable system you can for the least cost. "Efficient" is not always an intuitive or even obvious concept. With many years of experience assisting customers and teaching students, I have strong personal opinions about what constitutes efficiency.

---

8051 while adding additional features. This is the same strategy they used very successfully with the 8086, 286, 386, 486, and Pentium to capitalize on the installed software base. Philips has also introduced the 16-bit "upward-compatible" 80C51XA which allows existing 8051 code to be translated into code for the new processor. They both acknowledge the huge installed base of 8051 devices and are trying to provide a smooth migration path for more demanding applications.

For *low-end* (very high-volume, low speed) applications, there is still a place for 4-bit controllers such as National's COPS series. This is the very cost-conscious area where the choice of a processor is decided by pennies of cost difference at the 100,000-piece level!

<sup>7</sup>You can see that accessing any off-chip variable requires several instructions to set up the accumulator and data pointer.

**EFFICIENCY TIPS:** You will find highlighted sections discussing tips for efficiency—not just in code design, but also in hardware-software tradeoffs and in hardware design.

Efficient software is easy-to-understand software that uses subroutines (functions, described in Chapter 7) rather than straight-line programming so code can be reused. It uses tables, interpolation, and simplified calculations using the smallest possible variables so code is faster and smaller with simpler math operations.

Efficient software development takes advantage of all the tools available to minimize the time and effort required by the developer. The Windows-based environment ( $\mu$ Vision) supplied with this book is one example of efficient development. As you make changes, it automatically recompiles or reassembles the new versions and carries through to the machine-ready form.<sup>8</sup>

*An efficient application of a microcontroller involves designing with the minimum of external hardware necessary to allow the software to keep up with all of its tasks.* If there is only one processor and it is quite busy, the efficient path might to add more hardware. Usually though, it is more efficient to put as much responsibility as possible on software. That is where good use of interrupts and timers comes into play. The concepts of multitasking are vital.<sup>9</sup> The simple scheduler is a good beginning.

## REVIEW AND BEYOND

1. What are five programming languages used with the 8051?
2. Why is the 8051 an appropriate microcontroller for study?
3. From your own experience, give several examples of embedded controllers in consumer products.
4. Why does this book refer so often to the 8051 family?

---

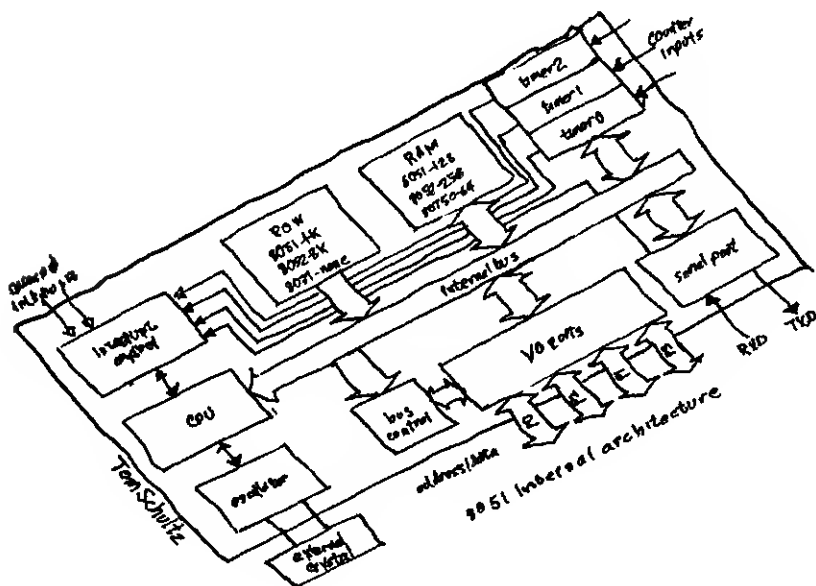
<sup>8</sup>Efficient development also means modular programming (Chapter 8) where you can easily reuse code as new assignments come along.

<sup>9</sup>The companion book goes into multitasking in much more detail and develops several systems from the code level.

# 2

## Computer Basics

---



### OVERVIEW

Many younger programmers have worked with high-level languages and advanced programming concepts without having any idea how the electronic hardware really functions. This chapter should give you a good feel for the underlying hardware. Some of this may be review, but understanding how the assembly instructions arise out of computer hardware will help you understand *why* the machine instructions of the next chapter are as they are.

## DISSECTING C: BACK TO BASICS

This is largely a book about the C programming language. Throughout it are C programs that somehow make things happen with an 8051 microcontroller. Magic? Not really. Here you will see how to work backwards from a C program to the specific machine instructions that run on the computer.<sup>1</sup>

Right at the beginning, I want to take you underneath the C program so you can get a picture of what is happening. To do so, I will start with a simple program used at Purdue in beginning micro classes. To run the program, students wire eight switches to one input/output (I/O) port and wire eight light-emitting diodes (LEDs) to a second port. The program reads in the setting of the switches, adds three to the value it read in, and sends the result out to the LEDs. I will talk more about hardware later—for now I want to talk about how the *software* comes into being and what it does.

---

```
/*Program to add three to switch inputs*/
#include <reg51.h>
#define lights P3
#define switches P1
void main(void){
    while (1){
        lights = switches + 3;
    }
}
```

---

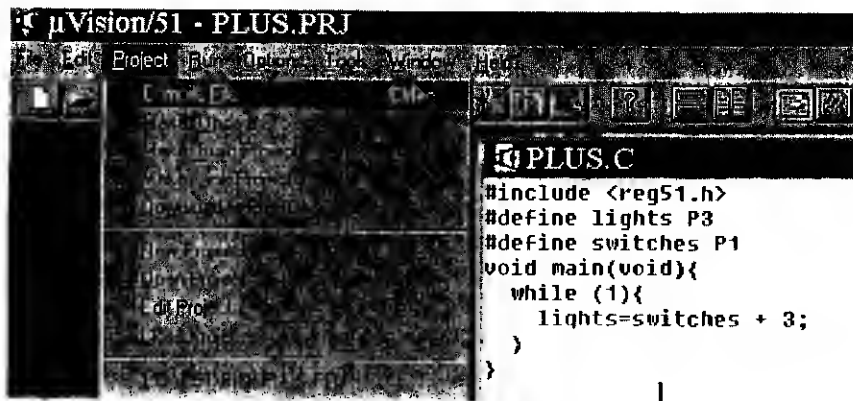
Here is a C program to do the job. As with most C programs, there are many extra lines before the working instructions. Actually, the *only* working code here is in the seventh line where the input from the switches has three added to it before the computer sends it out to the lights. The second line calls in a file that tells the C compiler all about the 8051-specific hardware—the designers of ANSI-standard C tried to avoid references to particular computer hardware. The next two lines assign the name *lights* to port 3 (*P3*) and the name *switches* to port 1 (*P1*). The line after that occurs because a complete C program requires a *main* function—later we will get into subroutines and functions. The *while(1)* is a way to say we want to do the operation over and over endlessly. The braces, { and }, are like the bookends to

---

<sup>1</sup>The machine instructions themselves are all discussed in the next chapter.

keep everything in place and show where things start and end. They are part of what makes C a *structured* language.<sup>2</sup>

So, using the development environment supplied by Keil, having typed in the program as shown, I asked to have the program compiled by clicking *Project, Compile file*.<sup>3</sup>



### Compiling the program

A window appeared, reporting that compiling was happening, and, about ten seconds later, it reported that the compilation was successful.<sup>4</sup> If not, it would have pointed out the program lines where problems were found and I could have fixed them and tried again. Being human, most of my software needs debugging before it is satisfactory to the compiler, and that is only the beginning of being sure it does what I want.

Next I clicked under *File, New* and found the list file (here *plus.lst*).<sup>5</sup> I have copied out only the essential parts.

<sup>2</sup>Any structured language has a number of rules, but the basic idea is that you enter a block of software at only one point and leave the same way—you do not jump into the middle of anything.

<sup>3</sup>The details of the development environment are discussed in Chapter 9.

<sup>4</sup>If I tell you which computer I used, fast as it may be today, you will laugh when you read it tomorrow. I will say that the tools are quite efficient compared to Microsoft's usual packages and that even a 386 does an adequate job with small programs.

<sup>5</sup>The listing shows the machine code because I set a parameter to have the code listed. Otherwise, it would only show the C-language instructions. See the section on using the environment for more details on these sorts of choices.

---

**INSTRUCTIONS RESULTING FROM COMPILING**


---

0000	E590	MOV	A, P1
0002	2403	ADD	A, #03H
0004	F5B0	MOV	P3, A
0006	80F8	SJMP	?C0001

---

On the right are the alphanumeric equivalents of the instructions, called assembly *mnemonics* (assembly language), which are easier for humans to understand. Remember that all we did was say we wanted to add three to the switches. The compiler did all the rest. It chose to first move the input from *P1* (the switches) to *A* (the accumulator—a temporary holding place). Next, it added the number three to *A*. Then it sent the value in *A* out to *P3* (the lights). Finally, it jumped back to read the input again in an endless loop. You could have written this assembly language program yourself.

On the left is the information that makes sense to the computer—the *machine code* or machine instructions. The first four digits are the address where the instruction is located. At address  $0000_{16}$  is an  $E5_{16}$ <sup>6</sup>—the machine code for a move into the accumulator from a specific memory location.<sup>7</sup> At address  $0001_{16}$  is a  $90_{16}$ —the address for *P1*. Together those two bytes make up one instruction. How the hardware does that move is discussed soon. The next chapter describes *all* the machine instructions of the 8051 family.

The second instruction is at address  $0002_{16}$ . The code  $24_{16}$  tells the computer to add a fixed value to the accumulator. The second byte of the instruction is the value to add—here  $03_{16}$ . Later in this chapter, I describe how the hardware does this arithmetic.

The third instruction,  $F5_{16}$  at address  $0004_{16}$ , moves the accumulator to an address found in the next byte—here  $B0_{16}$ , the address for *P3*.

The final instruction is the one to make this program an endless loop.<sup>8</sup> The short jump, code  $80_{16}$ , uses its second byte, with the address where the

---

<sup>6</sup>For a description of base 16 (hexadecimal) notation, see page 14.

<sup>7</sup>You can look up these codes yourself from Appendix A1. It is a good exercise to do a little of this by hand just to be sure you understand what the compiler and assembler are really doing.

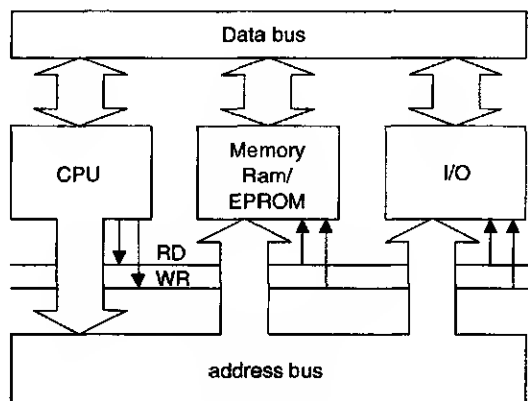
<sup>8</sup>Virtually all embedded controller programs are endless loops. After all, if the dedicated controller finishes its job, what is it supposed to do instead?

*next* instruction would otherwise be  $(0008_{16}$  in this case) to generate the new address  $(0008_{16} + F8_{16} = 0000_{16})$ . The *short* jump can only go to an address no more than 128 either way from the current location. It uses less instruction codes than a *long* jump that can go anywhere in the 16-bit address range of the 8051 family. Shortly I will discuss the way the instruction codes are brought into the computer and the way the instruction pointer changes.

## C or Assembly?

I think C is the best choice for programming the 8051, but if you want to understand the underlying operations, this is the place to start. If you are writing in C, you may happily skip to Section II on programming and will use this chapter only for reference when you have to dissect the results of your instructions, as I did here. If you write in C, a *compiler* will get you from high-level language to machine codes. If you write in assembly language, an *assembler* will get you from mnemonics to the equivalent numeric codes. You might be asked to hand-assemble by looking up the codes yourself, but that is something you should have to do only once or twice!

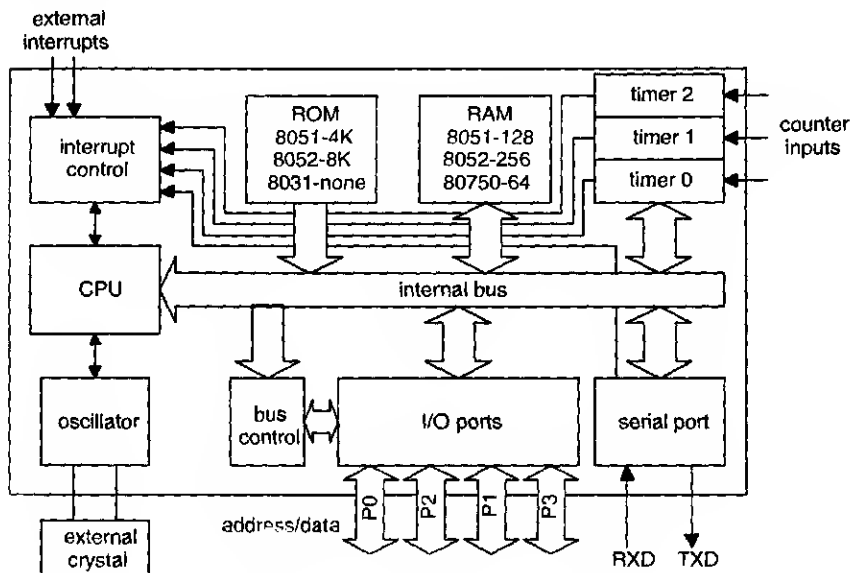
## COMPUTER ARCHITECTURE



Basic computer architecture



Before I describe machine instructions in detail (in the next chapter), I need to back up and introduce computer architecture in general and that of the 8051 family in particular. The previous figure shows the main parts of every ordinary digital computer.<sup>9</sup> The *central processing unit (CPU)* controls the activity. The instructions and data travel back and forth from the CPU to *memory* over the *data bus*. Communication with the outside world takes place through the *I/O ports*. The CPU's control of the transfer of instructions and data is by the *address bus* and the *read (RD)* and *write (WR)* lines.



**8051 Family internal architecture<sup>10</sup>**

A more detailed view, specific to the 8051-family chips, is shown above. Along with the CPU, there is the same *ROM* and *RAM* memory and *I/O*.<sup>11</sup> I show a single bus although the address and data almost certainly

<sup>9</sup>If you wonder about computers that do *not* have this architecture, investigate *analog* computers that can do certain types of computation very quickly, or investigate *neural nets*, which compute more like the human nervous system and brain. Most of the latter are experimental. The vast majority of “computers” from large office systems to Pentiums<sup>®</sup> to engine-control modules and traffic-light controllers all employ the basic architecture shown.

<sup>10</sup>The specifics of the individual family members differ—usually in the additional things they have—but the basic 8051 has all the above features except the third timer.

<sup>11</sup>Memory, bus, IO, and CPU discussions follow.

travel around internally on separate lines.<sup>12</sup> For now, ignore interrupts and timers.<sup>13</sup> In the next few pages, I put in more detail about the internal parts of the 8051-family. Do not panic if it seems to come too fast; it will come up again as the various instructions are discussed and is here only to help you understand some of the *why* of the instructions.

Think back to the first instruction of the sample program. It causes the CPU to issue the address of the port (from which data is to be fetched on the address bus), issue the signal to have the port's data put on the data bus, and then drive latches to hold the contents of the data bus in an internal register. There you have the steps for executing *MOVA, P1*.

The next instruction, *ADD A, #03H*, happens entirely inside the CPU. Well, not entirely inside because the instruction code has to be fetched from memory, but the actual process of adding of three uses one of the math instructions making use of the *arithmetic logic unit (ALU)* within the CPU. Besides addition, there are built-in instructions for subtraction, multiplication, and division, as well as logic operations such as AND, OR, exclusive-or, and complement. The hardware for all of this consists of digital gates. As we go along, you will start to understand how a CPU works.

In addition to reading data from I/O, the instruction bytes (*code*) have to come from memory, so I will describe that sequence as well. The following pages show how it is that digital logic recognizes addresses, puts data onto busses, and reads data off the bus to put in a register.

## Binary Numbers

By the way, if you have not encountered the terms yet, a single piece of digital information—a one or a zero—is called a *bit*. Four bits make up a *nibble* (or is it nybble?) and eight bits make up a *byte*. In representing binary numbers on paper, the right-most position is the least significant bit (*lsb*—with a weight of 1), and the left-most position is the most significant bit (*msb*—with a weight of 8 for 4-bit numbers,  $128_{10}$  for 8-bit numbers).<sup>14</sup> Sixteen-bit numbers are called *integers* in C or *words* in some other languages and hold up to  $65,535_{10}$ .

---

<sup>12</sup>I have no personal acquaintance with the actual internal design of 8051 chips, but later when I discuss multiplexed address/data lines you will see that it is much better to keep busses separate unless you are trying to save pins around the perimeter of the chip.

<sup>13</sup>Timers and interrupts are discussed much later (Chapter 11), but are vital to Multitasking.

<sup>14</sup>When looking at a binary number, you can get the decimal value by remembering that the weights, from right to left go 1, 2, 4, 8, 16, 32, 64, 128 and so on and adding in that value in each place where there is a 1. Thus  $1011_2$  is  $1 + 2 + 8 = 11_{10}$ .

Since it is very cumbersome to write binary numbers, groups of four bits are represented in *hexadecimal* notation.<sup>15</sup> The counting goes the same as decimal from 0 through 9, but it includes A, B, C, D, E, and F for the values 10 through 15. Thus  $1011_2$  is  $11_{10}$  or  $0B_{16}$ . Hexadecimal notation allows 8-bit numbers to be represented as two digits—00 through FF. In assembly language, hexadecimal numbers are represented with a trailing H, as in FFH or fffh, or in the C language with a leading 0x, as in 0xff.<sup>16</sup>

## Address Decoding

One of the steps to bringing in an instruction code or byte of data from memory (or a switch reading from a port) is having the proper device respond. It is like dialing a telephone number to cause a particular telephone to ring. When the CPU puts out an address on the address bus, some hardware has to recognize the address and make a particular memory location return its data to the CPU. This requires *address decoding*. The memory location must respond only when a specific combination of bits is on the address bus. You can do address decoding with common digital circuitry.<sup>17</sup>

## Single-address Decoder

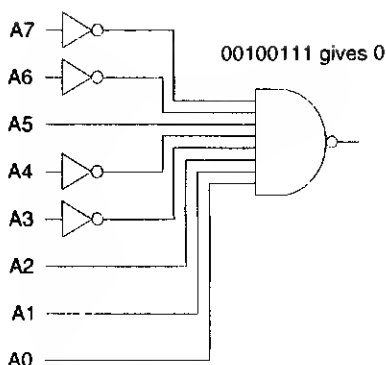
By taking a multi-input NAND gate and putting inverters ahead of specific inputs, you can make an address decoder that will give a low for just one combination of inputs. The circuit on the following page enables the output (logic 0) when the specific address  $27_{16}$  appears. Since a NAND gate itself gives a *zero* only when *all* the inputs are *one*, the zero out indicates that the particular address is there. The signal at the input to the NAND gate must be  $1111111_2$ , so here, with the inverters,  $00100111_2$  must be coming into the total circuit—an address of  $27_{16}$ .

---

<sup>15</sup>Hexadecimal for base 16—6 [hex] beyond decimal [10]. The number itself is still stored in binary as a set of bits—the notation is just a shorthand. Your instructor may quit in despair if you ask how to convert a number in the computer from binary to hex!

<sup>16</sup>A constant in C can be specified with a trailing U (unsigned) and/or L (long—32 bit). A now seldom-used notation is *octal* where each digit represents three binary bits. All the numbers look like decimal numbers (no letters), but they go only 0 through 7. Thus  $255_{10} = ff_{16} = 1111111_2 = 377_8$ . In C actual numbers must have a leading 0 (zero).

<sup>17</sup>I will assume you know digital logic symbols and truth tables. If not, you may grasp a bit less of the hardware basics but should be at no disadvantage in programming.



**Fixed decode for address 27  
using 8-input NAND and inverters**

The secret to understanding a decoder is to work backward from the select output—if *this output is low*, what must the inputs have been? In addition, if the inputs are that way, what does that make the address?

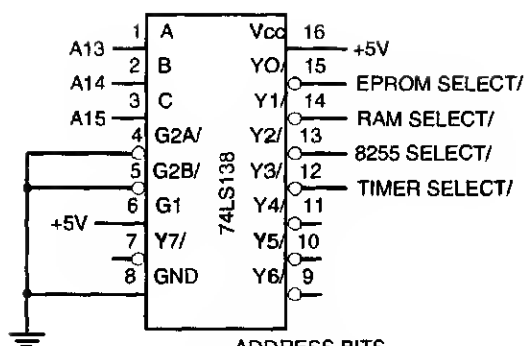
### Multidevice Address Decoder

Another common device for address decoding is the 74138. Given three input lines, it selects one of eight possible outputs. The truth table and schematic are shown on the next page.<sup>18</sup>

With this device, using six of the higher address lines, you can select among several different devices—several RAM or EPROM chips, latches for additional ports, or various programmable peripheral devices such as counters or LCD displays.

In the schematic on the next page, the 74138 has been wired to select an EPROM at 0000, a RAM chip at 2000<sub>16</sub>, a parallel port chip (8255) at 4000<sub>16</sub>, and a timer chip (8253) at 6000<sub>16</sub>. As you can see from the binary bit indications below the schematic, the circuit decodes only the top three address lines. For 8K RAM or EPROM devices, all the rest of the lines go to

<sup>18</sup>If you have not taken a digital course, some details about the schematic symbol may help. The small circles and the divide slash (/) indicate *negation*—the *true* condition for *negative* logic is a logic 0 coming in or going out. For example, the G2A signal is true when it comes in as logic zero. Likewise, an output is *asserted* or true when it comes out as logic zero. The truth table is the real thing to rely on, but the circles and NOT symbols help remind the logic designer of these inversions.



ADDRESS BITS

	A15	A8	A7	A0
EPROM	0 0 0	X X X X X	X X X X X X X X	
RAM	0 0 1	X X X X X	X X X X X X X X	
8255	0 1 0	X X X X X	X X X X X X X X	
TIMER	0 1 1	X X X X X	X X X X X X X X	

### Multidevice address decoder

the device itself. For the 8255, it happens there are only two address lines used in the chip (A0 and A1) so the rest are *don't cares*. If you think of the binary addresses of the chip, they are 010xxxxxxxxxxx00<sub>2</sub> through 010xxxxxxxxxxx11<sub>2</sub>. This means that the four addresses can be 4000<sub>16</sub> to 4003<sub>16</sub>, or they could be 4004<sub>16</sub> to 4007<sub>16</sub>, or any similar group of four ad-

74138 Truth table

inputs												
enb/s		se/sct			outputs							
G1	G2	C	S	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

Both G2s must be 0 to enable.

dresses up through  $5FFC_{16}$  to  $5FFF_{16}$ . When some of the address lines are not involved in selecting a device, the device is *not fully decoded*, and the repeated addresses are *fold-over addresses*.

**ADDRESS DECODING SHORTCUTS:** Although full address decoding with a chip such as a 74138 is a good idea, for small systems it is possible to avoid the extra chip by using the individual upper address lines. For example, one chip can be selected when A15 is low while another is selected when A14 is low. As long as you don't inadvertently address with *both* lines low, there is no problem. The potential problem is that you *can* address both devices at once which, if you are reading in, can cause bus contention. That won't probably damage the chips, but it can give strange results. A general suggestion is that you add a decoder only if you have more than two or three memory-mapped devices to select.

## PLDs

Another way of decoding addresses is with a programmable logic device (PLD). You can program such a chip to be a very custom logic device. It can behave as the 74138 just discussed, or it can select the EPROM and RAM but select the 8255 or timer much more specifically. Then the 8255 addresses might be *only* at addresses  $4000_{16}$  to  $4003_{16}$  and the timer *only* at  $4004_{16}$  to  $400C_{16}$ .<sup>19</sup>

## Memory

In addition to address decoding, a computer needs circuitry to hold instruction codes and data. In the beginning example of this chapter, the program had to store the switch settings it brought in before it could add three to them. The following sections describe the common storage components of microcontroller systems.

---

<sup>19</sup>The whole area of programmable logic devices is outside the scope of this book, but all new digital electronics textbooks now cover it. Some PLDs were one-time-programmable, but the newer ones are reprogrammable. Most modern microcontroller board designs use these devices to reduce the chip count on dense boards. Since most of the circuitry is in the microcontroller or other complex chips, the PLDs handle the simple inversions, ANDing, and decoding that fits things together. It takes the role of *glue logic* because it holds the big chips together.

**Registers.** If you group eight D-type latches in parallel, you have a byte *register*.<sup>20</sup> With eight inputs, eight outputs, and a single enable line (all eight enable/clock lines wired together), any time the enable is driven low, the inputs will show up at the outputs. Once you take away the enable (sometimes called a *clock*), even if the input changes, the latched number stays the same. It will stay latched in the register until some later enable stores a new number. Before you do that, hopefully, you will have latched the stored number in another register (perhaps in regular memory). If you do not save the old number, you *overwrite* it with the new one, and the information is lost.

**Memory arrays.** A register stores a single byte. A memory chip is a large array of storage bytes with the necessary address decoding and ways to route the bytes in or out of the array. The data travels in or out over one set of pins on the chip while the address comes in over separate lines.<sup>21</sup> The memory devices used with the 8051 family are usually *8-bit (byte) wide*, meaning that eight storage bits are at each address in the chip.<sup>22</sup>

## ROM and RAM

There are two fundamentally different types of memory devices commonly used with microcontrollers. The latches we described make up registers and *random access memory (RAM)*.<sup>23</sup> You can look at the outputs (read) of this memory and put in new data (write). It is *volatile*, meaning that the information is lost if you remove power.

On the other hand, you cannot change data in *read only memory (ROM)*. The outputs never change. ROM is excellent for holding code—the

---

<sup>20</sup>I again assume you already know about flip-flops. They are bi-stable storage devices—they can at different times hold either a 1 or a 0. You can make them up from simple gates (no one does!).

<sup>21</sup>Some of the early memory devices had separate in and out busses, and some devices send both the address and the data on the same lines at different times—called multiplexing, discussed more later.

<sup>22</sup>Although everyone now thinks in terms of the number of *bytes* of storage, the part numbers refer to the number of *bits* in the device. Thus a 2764 has 64K *bits* but is an 8K *byte* device (8K × 8). For 8051 family devices with a 64K byte maximum program storage spaces, you can hold it all in a single 27512.

<sup>23</sup>*Random* access means you can read any location in the array just as quickly as any other. This is in contrast to *sequential* access memory such as disk or tape memory.

program instructions that never change and should still be there after shutting off power.<sup>24</sup>

**Erasable programmable read-only memory (EPROM)** is a step beyond ROM. You can quickly store your program in the ROM chip, but shining ultraviolet (UV) light for about fifteen minutes through a window onto the chip die erases the program and the chip can then be reused.<sup>25</sup>

**Electrically erasable programmable read-only memory (EEPROM or E<sup>2</sup>PROM)** uses a different technology so you can erase devices without the UV light.<sup>26</sup> The write and erase times are not as fast as the read times—measured in milliseconds rather than nanoseconds—but erasing is much faster with UV light.

A newer technology for EEPROMs, that may make UV-erased EPROMs obsolete, is called **FLASH memory**. It is different internally in a way that makes the chip much smaller (and cheaper to manufacture) than EEPROMs. You program it byte-by-byte, but you erase it electrically in *blocks* of locations—not one address at a time. You can't use it exactly like RAM because you can't program an address more than once without first erasing, but it is excellent for holding programs or building up a block of stored data for a data acquisition system.<sup>27</sup>

---

<sup>24</sup>If you are mass-producing a product with an embedded controller, you might, after the bugs are out of the program, have the program built into the controller's *factory-masked* ROM. The manufacturer changes one step in the processing so the ROM part has the combination of 1s and 0s for *your* program. This process takes a number of weeks, costs many thousands of dollars to set up, and makes economic sense only when the program is sure and when the volume of devices you want is in the thousands, at least. It does also make it possible to keep your program secret from competitors.

<sup>25</sup>Programmable read-only memory (PROM) used to be common for digital logic where fusible links were actually melted inside the device to set up the permanent patterns. It was never a part of microcontrollers, although many controllers are now put out in one-time programmable (OTP) packages to save the expense of the clear-erase window. Technically they are PROMs rather than EPROMs because you can't get the light to the chips to erase them, but they are EPROM technology.

<sup>26</sup>In both EPROMs and EEPROMs the mechanism of storage is the trapping of charge in some storage locations and none in others. The charge leakage is so low that it stays there for at least decades—no one seems to worry about a hundred years from now because anything built today will probably be junked long before then!

<sup>27</sup>Like an EPROM, the programming goes one way—to logic 0, usually. If you are changing from one stored value to another that keeps all the 0s of the first binary number, you *could* program the new value without erasing first. The option is interesting, but usually not useful.



Finally, a competitor in the nonvolatile memory field is **battery-backed RAM**. First with separate circuitry and now right in the package with the chip, a small lithium battery provides the power to keep the volatile memory from ever losing power. You can write to any given location as fast as ordinary RAM, and values remain when you remove power.<sup>28</sup>

For completeness, even though it is almost never used with microcontrollers, I should mention **dynamic RAM**.<sup>29</sup> It is unusual to use dynamic memory with microcontrollers because the cost and space savings are not worth the complexity of refreshing. All the RAM needed in a microcontroller system can be obtained in one chip.<sup>30</sup>

## 8051 Code Storage

Code is typically stored in ROM since a program should never modify its own code and the code should be there whenever you turn the device on.<sup>31</sup> The program storage of the 8051 family is one of the main things that differentiate the members. A few members of the family hold the original 2K bytes, while one holds only 1K and some hold as much as 32K. Some hold no on-chip code and must have external code storage, as will be described near the end of this chapter. There are both factory-masked and EPROM versions, and at least one company, Atmel, has gone to FLASH memory for code. A table in appendix A5 lists these differences.

---

<sup>28</sup>Battery-backed RAM devices are a bit more expensive than EPROMs, but they are very useful for data acquisition systems. There is some protection in the slow-write devices though—if something goes wrong, it probably happens too fast to trash any significant amount of memory, whereas the RAM could be really messed up. I believe the battery-backed devices are guaranteed to hold their data for at least ten years—“forever” for most of today’s computer electronics.

<sup>29</sup>There are two basic types of RAM—*static* and *dynamic*. Static RAM requires at least a two-transistor arrangement for each bit. The state of the bit is permanent as long it has power. Dynamic RAM relies on charge stored in a *single* transistor acting like a capacitor. The charge slowly “leaks out,” so you have to periodically read the level of charge and write it back to full or empty as the charge drains off. An advantage of digital electronics is that everything is either 1 or 0, so as long as this refreshing takes place before too much charge has drained off, the circuitry “knows” whether to fill up the charge or remove it. This preserves the logic state. If you delay the refreshing too long, you lose the correct value. Both types of RAM are *volatile* in that the memory goes away if you remove power.

<sup>30</sup>PCs, needing 10s of Meg of RAM, use dynamic memory except for the smaller cache memory, where the faster speed of static memory is used to buffer between the very fast processor and the slow dynamic memory.

<sup>31</sup>Even a PC has some ROMed code—the bootstrap ROM and the BIOS that give the computer enough program to go out to the disk to get the “real” program.

## 8051 Internal RAM

Within the 8051 family, there can be anywhere from 64 to 256 bytes of *internal RAM* to hold data. With the 8051 family, this all-static memory is the fastest and most varied in addressing modes.

*Special function registers (SFRs)* are internal 8051 memory locations that control the on-chip “peripherals” such as the ports, timers, and interrupts, as well as other features of the processor. In the beginning example with the lights and switches, both ports are special function registers (*P1* at address  $90_{16}$  and *P3* at  $B0_{16}$ ). Other SFRs of particular interest are the accumulator (*ACC*, at address  $E0_{16}$ ), the *B* register (at  $F0_{16}$ ), and the data pointer (*DPH* at  $83_{16}$  and *DPL* at  $82_{16}$ ). The table here shows all the SFRs for the standard 8051. Appendix A5 shows additional SFRs that exist in other family members. They all figure prominently in the instructions.<sup>32</sup>

**Special function registers for 8051**

Symbol	Description	Direct Address
<i>P0</i> <sup>1</sup>	<i>Port 0</i>	$80_{16}$
SP	Stack pointer	$81_{16}$
DPTR	Data pointer (2 bytes)	
DPL	Data pointer low	$82_{16}$
DPH	Data pointer high	$83_{16}$
PCON	Power control	$87_{16}$
<i>TCON</i>	<i>Timer control</i>	$88_{16}$
TMOD	Timer mode	$89_{16}$
TL0	Timer low 0	$8A_{16}$
TL1	Timer low 1	$8B_{16}$
TH0	Timer high 0	$8C_{16}$
TH1	Timer high 1	$8D_{16}$
<i>P1</i>	<i>Port 1</i>	$90_{16}$

<sup>1</sup>Registers in italics are also bit-addressable, as will be discussed with the bit instructions.

## 8051 Off-chip Memory

With the exception of a few of the smallest members, all the 8051 family can access off-chip memory space. To do so, the upper 8 bits of the address bus take over I/O port *P2*. The lower address and the data together take over port *P0*. To save pins—the original 8051 has only forty of them—port *P0* first

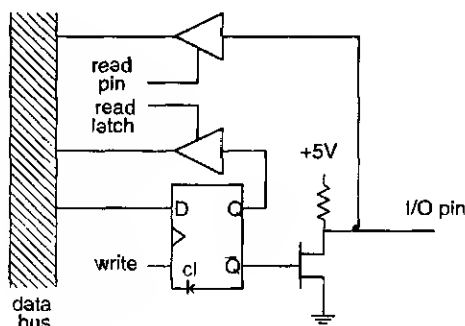
<sup>32</sup>If you issue a direct address instruction to an address from  $80_{16}$  through  $FF_{16}$ , you are addressing special function registers. For example, the *ACC* register (byte address  $E0_{16}$ ) can be bit addressed from  $E0_{16}$  (lsb) through  $E7_{16}$  (msb).

serves as the lower 8 bits of the address bus and then as the data bus. This is **multiplexing**. It saves pins but reduces the speed of memory access and requires an off-chip address latch.<sup>33</sup> External access with the 8051 is always slower because of the multiplexing as well as because it takes one extra instruction to set the data pointer before the instruction to actually access the off-chip location.<sup>34</sup> There are only three instructions for such access.<sup>35</sup>

## Input/Output (I/O)

Without some way to exchange information with the “outside world,” a computer is worthless. While PCs have fairly standardized input/output connections (*COM1*, *LPT1*, and so on), microcontrollers excel in providing much more adaptable input/outputs. The term **port** is used to refer to a block of I/O.<sup>36</sup> There are two common types of ports—serial and parallel.

**Parallel ports.** Parallel ports are groups of (usually eight) bits on individual pins. A parallel *output* port latches its value until you send out a



**Basic internal port pin**

<sup>33</sup>See the later section on control bus signals for details of the address latching.

<sup>34</sup>For example, to write the contents of the accumulator to off-chip location 307B<sub>16</sub>, you would need two instructions—*MOV DPTR,307BH* and *MOVX DPTR,ACC*.

While some of the newer relatives have instituted additional pointers and one Dallas chip, I believe, has put “off-chip” RAM on-chip *without the multiplexing*, for most 8051 members it is a very slow process to work in off-chip memory.

<sup>35</sup>Be careful to distinguish between code and data space. While code can be on or off the chip just like data, there are no instructions to write *to* code space.

<sup>36</sup>The term *port* in computers perhaps arose from the parallel with the way goods go in or out of a country by the ports. Things go on inside the country or chip, but the travel to the outside always goes through a port.

new value. You can make off-chip parallel *output* ports with 8-bit latches. A parallel *input* port passes the states of the pins into the computer at the instant that the move instruction for that address executes. You can make parallel input ports from octal tri-state buffers.

The 8051 connection to the outside world is usually through ports. There are always a few port pins available directly on the chip and there can be additional ports added on the expansion bus, discussed later.<sup>37</sup> Internal ports of the 8051 are unusual in that they are only *partly* bi-directional (the term is *quasi*-bi-directional). You can see the hardware in the schematic for one pin on the previous page. You can drive *out* any time but, for *input*, the output circuitry must be off. Otherwise you will be reading your own output latch rather than the signal coming in from the outside.<sup>38</sup> The key software aspect is the requirement that *any port you are using as input must have a 1 written out to it*.<sup>39</sup> There are other functions for some of the port pins not shown here. Suffice it to say, if you are using a pin for other functions you should not use it as a port. There is no special disabling or setting of port direction.<sup>40</sup> Remember that the port pins can sink about 1.6mA but only source tens of  $\mu$ A. Also, P0 when used as an I/O port (as opposed to the data bus for off-chip memory expansion) does not have a pull-up resistor—the external device must supply the logic high.

**Serial ports.** Serial ports transfer single bits of data one after another, taking at least eight transfers to exchange a byte. The most common form of serial port uses just one pin for each transfer direction.<sup>41</sup> Usually on-chip hardware handles the timing details and includes parallel/serial shift

---

<sup>37</sup>The 8051 family is different from the x86 devices in that all I/O is memory mapped—there is really no difference between accessing memory and accessing I/O. This is different from the x86 family where there is a different set of instructions for I/O than for memory access.

<sup>38</sup>When two logic outputs are tied together, if there is a “winner” at all, it is the one pulling low to zero.

<sup>39</sup>On reset the latch is set high, so if you never write out, you do not need to worry. If you write out a 0, when you read in the port you will see the 0 being sent out rather than the condition of the external device.

<sup>40</sup>Input operations read either the latch or the port pin, depending on the instruction. The instructions that use the latch when a port is designated are the read-modify-write instructions: *ANL*, *ORL*, *XRL*, *JBC*, *CPL*, *INC*, *DEC*, *DJNZ*, and three of the bit instructions. They should never have an *input* port as a destination for the result anyway. The distinction protects from reading a false state of an output port due to pin loading.

<sup>41</sup>*Asynchronous* transmission synchronizes on the receive end by recognizing a start bit. *Synchronous* transmission does not need the extra bits but requires that data be sent all the time to keep the receiver in sync. It is much less common.

register circuits. As a whole, the hardware for this function is a universal asynchronous receiver/transmitter (UART). It relieves the processor of the job of managing the timing and organizing of the bits so the processor can do other things. It is possible to send serial information directly over ordinary port pins using software and timers—particularly when the transmission rate is low—but it is better to use the UART hardware when possible.<sup>42</sup>

## Moving Data—Busses

In the example at the start of this chapter, the software brings the switch readings to a storage location inside the microcontroller. To get there, the data from the switches has to travel over a *bus*—a group of wires carrying the parallel bits of a binary number or related signals.<sup>43</sup> A bus can carry eight bits of data,<sup>44</sup> a set of control signals such as *read* or *write*, or the bits of an address.<sup>45</sup> Although you may have encountered it in a digital logic class, to understand how a bus can move data, it is good to review tri-state logic.

**Tri-state logic.** This routes virtually all computer data.<sup>46</sup> In my far-off college days, I learned that binary devices have only two valid states—a 1 or a 0: *bi*-state if you wish. Some time after the birth of TTL, a “new” type of logic came along where the output could be 1, 0, or floating. The third state you can think of as *out of the picture; don't care; let some other device*

---

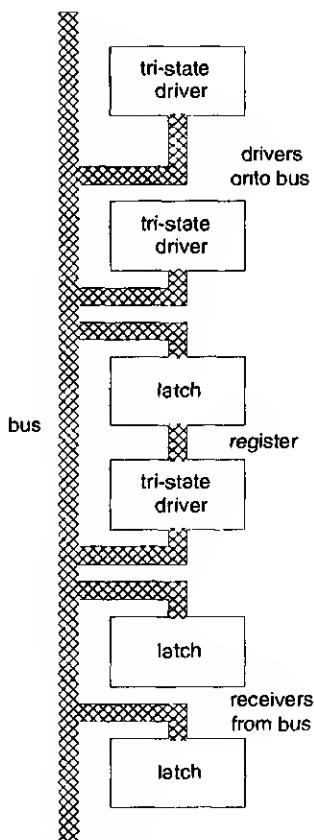
<sup>42</sup>A different serial transfer technique uses a separate clock line rather than synchronizing off the data. One form is now common for new peripheral devices such as AD and DA chips because it reduces the number of pins needed. The device that controls the clock line controls the data transfer rate. A more elaborate form of this technique, invented by Signetics (now Phillips), is called the I<sup>2</sup>C bus. It is discussed in detail in the companion book.

<sup>43</sup>Perhaps, in the usual haphazard way that engineering terms grew up, it is called a *bus* because different signals get on and go for a ride on the bus. Or it may have come from the term in electrical distribution applied to a heavy conductor for power distribution.

<sup>44</sup>The width of its *data* bus characterizes a micro. The data bus is 8-bits wide for an “8-bit micro.” Many processors have a wider internal data bus than the one that comes out to the off-chip devices—it is much easier to run wide busses internally than to come up with all the extra pins to bring them off the chip. With the trends in surface-mount ICs, that distinction is becoming less significant. For comparison, the 8086 microprocessor had a 16-bit data bus and 20 address bits; the 286 had 16 data bits and 24 address lines; the 386 and up have 32-bit data bus and 32-bit address bus.

<sup>45</sup>Addresses in the 8051 family, usually of memory storage locations or I/O devices, are 16 bits wide (65,535 possible addresses).

<sup>46</sup>There are non-tri-state data selecting logic devices such as the 74150 that route one of 16 inputs to the single output, but they have dropped out of general use.



### Bus made of tri-state drivers and latches

*determine the value.* While conventional logic *never* approves of tying *outputs* together, tri-state logic encourages just that. You can connect several—even many—device outputs to the same wire. As long as only one of the tied-together outputs is active at a time (the rest must be tri-stated/floating), the one active output controls the level of the wire.

**Building a bus.** For one bit of a bus, connect several *tri-state output* devices together—the senders. Then connect several *input* devices to the same wire. You now have a (1-bit) bus. Put eight bits in parallel and you have an 8-bit bus. You can stretch the connections out over a long distance or lump them closely together, but electrically you still have a bus. Supply the correct enable and latch signals to the connected devices and you can

route data onto the bus from any output and latch it into another device on the bus. This is the secret of all the 8051's data exchange with off-chip memory as well as most of the internal data transfers.

## 8051 Instruction Execution Sequences

Before getting into the machine instructions in the next chapter, you should have a general idea of the multistep process involved in executing an instruction.

The *fetching* of an instruction code requires first sending out the *address* for the instruction (from the program counter) by enabling a tri-state buffer in the CPU to drive it onto the address bus. When going to code storage, the low part of the address only stays until the ALE signal goes away since the same pins will handle the code byte.<sup>47</sup> During the time from when the address is on the bus, the external memory device holding the instruction can decode the address and recognize that it may be required to respond.

For a code fetch, after a delay controlled by the clock, the processor puts out a *program store enable* (*PSEN*) signal. This causes the code storage device to enable its tri-state outputs and to drive the data bus with the contents of the particular instruction location being fetched. At the end of the *PSEN* signal, once the code byte is on the bus, the CPU latches in the code from the data bus.<sup>48</sup>

Next the CPU *decodes* the instruction so the proper sequence of operations can be carried out—do a particular math operation, move a byte of data out to a specific location, compare two values, or even change the address of the next instruction fetch (a jump).<sup>49</sup>

---

<sup>47</sup>As mentioned earlier, to off-chip devices the address and data are multiplexed. The code access timing is the same even on-chip, however.

<sup>48</sup>As mentioned earlier in this chapter, the execution of an instruction first involves fetching the instruction from code memory. The processor fetches two code bytes although only one may be needed—it just ignores the second fetch if the code from the first fetch doesn't require a second byte.

<sup>49</sup>Once the processor brings the first code byte in, it is decoded. That decoding process is what makes 8051 the processor it is. Until recently all the microprocessors were complex instruction-set computers (CISC) like the 8051. Some instructions have a different number of bytes from others and take different amounts of time to execute. The multiply instruction, for example, involves a series of operations that would otherwise involve a whole series of instructions. You may want to look at the assembly example in Chapter 10 where multi-byte math is done, to get a feel for such a process. Some of the x86 instructions are also quite complex—particularly the string moves that can take many machine cycles to complete.

Lets take a specific instruction—say a command to move the contents of memory location 2045<sub>16</sub> into the data pointer register (*MOV DPTR, #2045H*). When the first byte of the instruction has been decoded, it shows that two more bytes of instruction are needed—the two bytes to go onto the register.<sup>50</sup>

The next step in executing this particular instruction, then, is to increment the program counter by one and fetch the second instruction byte. Finally, the third instruction byte is fetched. The incoming bytes in this case go directly to *DPTR*. A look ahead to the table of all instructions (Chapter 3, page 52) shows that this process takes 24 clock cycles for the *MOV DPTR* instruction.<sup>51</sup>

Consider an instruction that transfers data (as opposed to code bytes) to external devices. In this case, the *RD* or *WR* control lines come into play. Take the *move to external data* instruction (*MOVX*). It requires that the number already be in the accumulator (perhaps with a *MOVA, #DATA* command) and the external memory address already be in the data pointer (*DPTR*). The steps to executing the instruction (*MOVX @DPTR, A*) involve fetching the instruction byte, decoding it, putting the *DPTR* value out on the address bus, putting the accumulator value out on the data bus, and then issuing the *WR* (memory write) signal. That is the CPU's role. All the mem-

---

The opposite of CISC is the reduced instruction-set computer (RISC). These processors have far fewer and less complex instructions so it takes more instructions to get something done. In return, they execute the instructions much more quickly and involves less internal hardware. The jury is still out on the relative merits of both. The RISC machines are faster *per instruction*, but it takes more of them to do a job. Producing the instructions is a good job for a compiler so the C programmer does not even have to know about the underlying instruction set. The continually changing performance of the silicon itself and the variety of applications to consider obscures the architecture debates. It is about like asking if digital signal processing (DSP) chips are "faster." It all depends on what you are doing. Highly repetitive parallel operations common to DSP work well, but they would probably do a poor job on hardware control. I recently saw an advertisement claiming that a particularly fast CISC chip could do a job faster and more cheaply than a slower CISC chip combined with a separate DSP chip! Other processors would have different instruction decoding built in. The instruction decode then sets gates to determine what happens on subsequent clock edges. The instruction could indicate a direct action on the accumulator (*INC A*) or the preparation to retrieve another instruction code such as the address of a register (*MOV A, R7*) or a constant value (*MOVA, #25H*). The instruction could also set up the process of going off-chip to read or write external memory.

<sup>50</sup>Since an 8051-type processor has only eight data lines (an "8-bit processor") but 16 address lines, the 16-bit pointer address is fetched in two 8-bit installments.

<sup>51</sup>2 $\mu$ Sec with a 12MHz clock.



ory device does is recognize the address with its memory decoding circuitry and latch in the data when the *WR* signal ends.<sup>52</sup>

## TIMING AND SIGNAL DETAILS

### Control Bus Signals

Having outlined the instruction execution process, let us go into the signals in more detail. The signals *within* the 8051 are not described in the data books and aren't of much concern to you when using the devices. What are far more important are the external signals for expanding to off-chip memory, adding ports, or interfacing to other devices. All these signals pertain to off-chip expansion.

### ALE

This signal, address latch enable (*ALE*), is for driving an external latch for capturing the low part of the address put out on *P0* before the data is transferred to or from external memory.<sup>53,54</sup>

---

<sup>52</sup>All 8051 instructions involve 12 or 24 clock cycles (except 48 for multiply). Fetching an instruction code can be done in 6 clock cycles, so 3-byte instructions stretch out to 24 overall. To allow for slow RAM and EPROM devices, access to off-chip memory is slowed to 24 clock cycles. The most useful information to be gathered is that most 8051 instructions take one or two microseconds. The clock cycles of instructions are useful also when you make a time delay by making a program loop.

There are 8051 family devices that can run at up to 16 or even 40MHz, so instructions can take less than a third of those times in some cases. Also some of the Dallas devices run fewer clock cycles per instruction so the processor speeds up with the same clock speed. These advanced features are probably fallout from the intense efforts to make the high-performance computers faster and more efficient by trading off circuit complexity for speed. The semiconductor technology of the 8051's introduction is a far cry from today's and the 8051 family still merits the development work to improve it. Witness the 251 from Intel being upward compatible down to the code level.

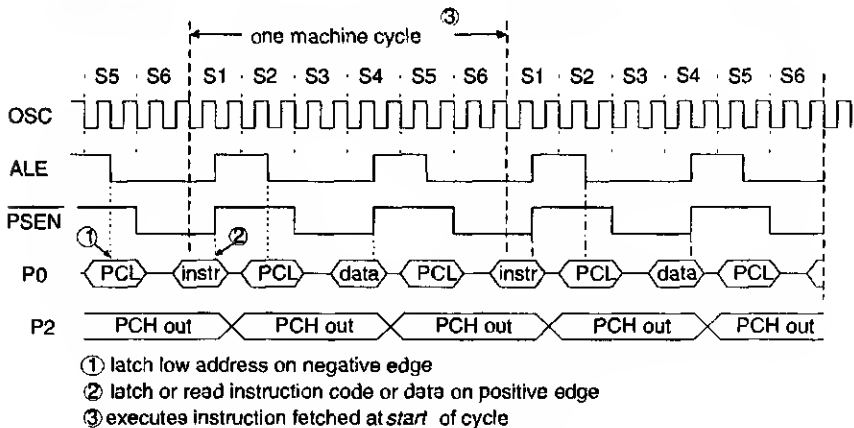
<sup>53</sup>*ALE* also has use as an "almost-clock." With the exception of times when external data space is addressed with *MOVX* or *MOVC*, *ALE* runs along quite regularly at 1/6 the clock frequency. When you need a sloppy clock (one where an occasional missing pulse won't matter) to run something like an A-D converter, this can often suffice. See the figure of the *MOVX* timing for details.

<sup>54</sup>There exist a few external memory and I/O chips that have a latch in the chip (Intel's 8256, and one or two port/memory chips designed for the 8085 microprocessor may still be available), but most designs use a separate 8-bit latch (usually a 74LS373) to hold the low address byte. In a special (seldom used) set of instructions, the off-chip address range (called *pdata* in

## PSEN

Program store enable/ (*PSEN*) indicates that the ROM should put code on the data bus. *PSEN* is only active if access is to *external* memory. If you tie *EA*/ pin to ground then *all* access is to off-chip ROM. If you leave *EA*/ to float high then accesses to the addresses within the on-chip range do not cause the *PSEN*/ to go low.<sup>55</sup>

**FORGETTING TO TIE EA LOW FOR OFF-CHIP CODE APPLICATIONS:** It is easy to ignore the "unimportant" pins when wiring up a micro, but in this case using an 8031 or 8032 with off-chip EPROM will give you a system that doesn't fetch any code.



### External code-fetch timing

Keil/Franklin C) covers only 256 locations and the address is put out only on *P0*. Software can then control port bits on *P2* (or any other port) to *page* the memory—in other words, you can switch between 256 byte pages of off-chip memory by a separate instruction to change a few port bits that feed directly into upper address bits of a RAM chip. The havoc this makes in variable assignment far outweighs the few port bits saved, and nothing is saved if off-chip code space is needed. Usually the full 16-bit address range is used.

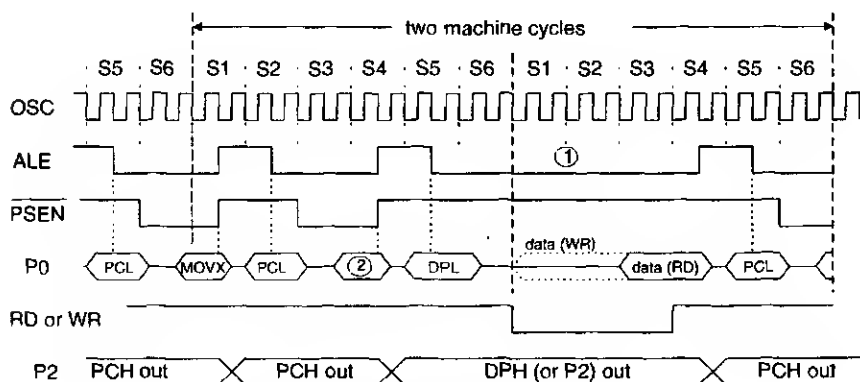
<sup>55</sup>Be aware that the ROMless versions of 8051-family chips must have the *EA* pin tied low to do anything—otherwise they still try to access nonexistent *internal* ROM on startup.

The waveforms in the figure on the previous page show off-chip *code* fetch (ROM) activity. The sloping edges suggest that there are rise and fall times associated with the signals while the halfway (neither 1 nor 0) signals for *P0* show that the bus is tri-stated in between. If you attach an oscilloscope to the bus lines, you may well see something much messier—especially during the tri-stated (float) times. Remember that a given microcontroller has a maximum speed because of these non-instantaneous transitions. The closer you run the chip to its upper speed limit, the more rounded the signals will look.

## RD and WR

These signals are *only* used for off-chip data storage devices and external ports.

The next waveforms show the timing for a *MOVX* instruction. If you use the *MOVX* instruction with the accumulator as the *destination*, you will issue a *read* signal, and, with the accumulator as the *source*, a *write* signal.<sup>56</sup> Showing a three-byte instruction reemphasizes the fact that all the instruc-



- ① note that *ALE* is only missing for *MOVX* instruction
- ② single byte instructions still have second fetch phase

### Off-chip read/write (*MOVX*) timing

<sup>56</sup>You have to give up use of one pin on *P3* if your program does any external reading. Likewise, a write will cost a second pin. There is no hardware setup for this internally—be careful only to *bit-address* *P3* if you have also added off-chip RAM. There are actually two *external* addressing modes—one uses *DPTR* (*xdata* in C) and the other uses *R0* or *R1* (*pdata* in C). In the latter case *P2* continues to function as an ordinary port while in the former case the processor puts out the high part of *DPTR* on *P2*.

tion fetch cycles involve two bytes so the one- or three-byte instruction fetches include a *dummy* instruction fetch.<sup>57</sup>

## The Oscillator

All of the 8051 family members use an external crystal for the oscillator.<sup>58</sup> You can, depending on the family member, choose a crystal frequency from 500 kHz to 33 MHz or 40 MHz. Check the specific data sheet for more details.<sup>59</sup>

## Clocks and Timing

When looking at how the processor actually carries out instructions, notice the basic machine cycle is *not* one clock period—each instruction does *not* happen in 83 nsec if you have a 12 MHz crystal. Instead, for most of the 8051 family it takes *12 clock cycles* to complete a simple instruction.<sup>60</sup>

Like virtually all digital computers, the entire operation of the 8051 family is *synchronous*—everything happens in step with the clock. There is a very consistent sequence to executing an instruction. In fact, some instructions have to “waste” time just to keep in step.<sup>61</sup>

---

<sup>57</sup>The Dallas chips overcome this inefficiency—the extra fetch cycles are eliminated. The chips are not *entirely* compatible since the timing of any software loops will not remain the same as with a conventional 8051 family member running the same program. If you have written a good application that derives its timing from hardware timers rather than software delays, you should have no problem.

<sup>58</sup>Check individual data sheets for other options such as external clocks or R-C oscillators—if you decide to go this way, note that the NMOS devices have the clock injected on *XTAL2* while CMOS devices use *XTAL1*.

<sup>59</sup>The most common frequency is 11.059 MHz because many devices only run up to 12 MHz. The slightly lower frequency permits one of the timers to generate the necessary clock frequency for the serial port to operate at 9600 Bd—at the high baud rate the divide is only by 3 so there isn't room to adjust for the 12 MHz rate. For very slow baud rates the divide is large enough to produce a good enough clock from almost any crystal frequency. See Chapter 11 for details.

<sup>60</sup>Review the last column of the instruction tables in Chapter 2 to get the specific count—a few take 24 clock cycles and the multiply and divide take 48 clock cycles. The only exception presently is the Dallas Semiconductor High Speed Microcontroller (HSM 520) chip family that complete many instructions in 4 clock cycles, which makes its members inherently up to three times faster than their clock frequency suggests. This is not universally the case for the Dallas 520 chips—some instructions take 8 instead of 12, some take 12 instead of 24, and the multiplies take 20 instead of 48. Dallas suggests the chips execute *typical* programs in half as many clock cycles.

<sup>61</sup>Again, the Dallas HSM chips save time by avoiding the extra fetch when it is not needed.

## Clock Cycles, Machine Cycles, and Instruction Timing

How soon must the data you supply have settled before the microcontroller's *RD* signal goes away? How long does the *WR* data remain after the *WR* signal goes away? These are the questions to ask when you are interfacing other devices directly to the external bus. See the specific processor data sheets for exact time relationships for the various signals. For connecting to most external devices there is no problem. Any TTL devices are immensely faster than most of the 8051 family. Today static RAM and EPROM are much faster than the micro as well.<sup>62</sup>

Timing the specific bus signals is only of concern when interfacing other devices, but the *overall* timing of the instructions can enter into your program development. First, if it takes too long to execute a series of instructions, the program may not get everything done before new things need doing. This is discussed much later in Section III (Multitasking).

A second reason to know instruction timing is if you produce *time delays* by purposely using instructions to take up time. You might want to issue pulses to a stepper motor at a regular interval—say 100/sec. The trick is to make a program loop (perhaps with a *DJNZ* instruction—a *for* or *while* loop in C) such that the program has taken up 10 msec when the looping finishes. To get an idea of the process, take the program below that will loop as long as it takes to get *A* counted down to zero and then go on. How long will it take?

---

Delay loop with machine codes			
0000	74FF	MOV A, #0FFH	
0002	14	LOOP: DEC A	
0003	70FD	JNZ LOOP	

---

To answer the question we can either run the program on the simulator (and let the simulator count the number of clock cycles) or, as an exercise, we can do the math.<sup>63</sup> A quick check of the instructions from the next chap-

---

<sup>62</sup>The only common device where there could be a speed problem is the alphanumeric LCD display module—it turns out that the specified *WR* (*enable*) pulse is longer than you will get from an 8051 at 12 MHz. It seems that this is not really a problem because I have seen numerous student projects interfacing to them with no apparent timing problem. There may be interface problems as you increase the crystal frequency or go to the Dallas chips. At 25 MHz, times are cut in half. With the Dallas chips there is a default mode where *MOVX* takes three of their machine cycles but can be set to take only two cycles. Thus the chip starts up compatible with the device it replaces but can be set to run faster with faster ROM and RAM. See their application notes if you find yourself in need of more information.

<sup>63</sup>Using the ds51 simulator the number of clock cycles turned out to be 766, but that includes the initializing of the accumulator.

ter indicates that *DEC A* takes 12 clock cycles to execute and *JNZ* takes 24 clock cycles. To a first approximation then, each number in the accumulator represents a delay of 36 clock cycles. With a 12 MHz clock that is 3  $\mu\text{sec}$  per loop or  $255 \times 3 = 765 \mu\text{sec}$ , so including this loop would produce about 1300 stepper motor steps per second. This is much quicker than the desired 100/sec so you might put the loop inside another loop. In Chapter 5 (pages 128–129), I go over this in more detail. The point here is that you can, if you must, calculate *exactly* how long it takes a program to execute.

## Register Banks

Part of the 8051 internal memory can be addressed as four *register banks*—groups of 8 bytes. The designations *R0...R7* refer to those eight bytes. The actual on-chip RAM locations of these registers can be one of four places, depending on the setting of two bits in the program status word (*PSW*).<sup>64</sup> Register banks allow very rapid moving from one activity to another.<sup>65</sup>

In assembly language, control of register banks is a matter of programming 2 bits in the *PSW*.<sup>66</sup> In C, the choice of register banks depends on specific compiler directives. In Keil/Franklin C you can use the *registerbank* directive to choose a bank for the entire program module, or use the *using* directive to do the same for a single function.

## Special Function Registers

The special function registers (SFRs) control the peripherals on the chip. The table on the next page lists all the special function registers of the 8051.<sup>67</sup> Specifics of each register are covered with the related internal peripheral.

The ports and several other SFRs are also bit-addressable so, for example, *P3* can be addressed as a byte at  $B0_{16}$  or it can be addressed with bit

---

<sup>64</sup>The setting of those bits directs references to *R0–R7* to internal RAM address 00–07 (00<sub>2</sub>—registerbank 0), 08–A<sub>16</sub> (01<sub>2</sub>—registerbank 1), 10<sub>16</sub>–18<sub>16</sub> (10<sub>2</sub>—registerbank 2), or 18<sub>16</sub>–1F<sub>16</sub> (11<sub>2</sub>—registerbank 3).

<sup>65</sup>Here the change of 2 bits can save all 8 registers. References to registers will not go to the same 8 bytes until you restore the two *PSW* bits. The alternative is pushing and popping to a stack (normal with other processors). This is the subject of much more discussion in Section III where it relates to context switching.

<sup>66</sup>If you are linking modules with mixed language programming, you can specify the banks used in the assembly program so the linker will not use the bank area as ordinary memory.

<sup>67</sup>Some family members have many more SFRs for control of the additional features and a few members have fewer where ports or timers are not part of the design.

instructions at addresses  $B0_{16}$  through  $B7_{16}$ . Registers that are also bit-addressable are shown in italics in the table. All the SFR byte and bit addresses are above  $7f_{16}$  to avoid conflict with the on-chip RAM (directly addressable) and the bit memory.<sup>68</sup>

**Special function registers for 8051<sup>1</sup>**

Symbol	Description	Direct Address
<i>P0</i>	<i>Port 0</i>	<i>80<sub>16</sub></i>
<i>SP</i>	<i>Stack pointer</i>	<i>81<sub>16</sub></i>
<i>OPTR</i>	<i>Data pointer (2 bytes)</i>	
<i>DPL</i>	<i>Data pointer low</i>	<i>82<sub>16</sub></i>
<i>DPH</i>	<i>Data pointer high</i>	<i>83<sub>16</sub></i>
<i>PCON</i>	<i>Power control</i>	<i>87<sub>16</sub></i>
<i>TCON</i>	<i>Timer control</i>	<i>88<sub>16</sub></i>
<i>TMOD</i>	<i>Timer mode</i>	<i>89<sub>16</sub></i>
<i>TL0</i>	<i>Timer low 0</i>	<i>8A<sub>16</sub></i>
<i>TL1</i>	<i>Timer low 1</i>	<i>8B<sub>16</sub></i>
<i>TH0</i>	<i>Timer high 0</i>	<i>8C<sub>16</sub></i>
<i>TH1</i>	<i>Timer high 1</i>	<i>8D<sub>16</sub></i>
<i>P1</i>	<i>Port 1</i>	<i>90<sub>16</sub></i>
<i>SCON</i>	<i>Serial controller</i>	<i>98<sub>16</sub></i>
<i>SBUF</i>	<i>Serial data buffer</i>	<i>99<sub>16</sub></i>
<i>P2</i>	<i>Port 2</i>	<i>A0<sub>16</sub></i>
<i>IE</i>	<i>Interrupt enable</i>	<i>A8<sub>16</sub></i>
<i>P3</i>	<i>Port 3</i>	<i>B0<sub>16</sub></i>
<i>IP</i>	<i>Interrupt priority</i>	<i>B8<sub>16</sub></i>
<i>PSW</i>	<i>Program status word</i>	<i>D0<sub>16</sub></i>
<i>ACC</i>	<i>Accumulator</i>	<i>E0<sub>16</sub></i>
<i>B</i>	<i>B register</i>	<i>F0<sub>16</sub></i>

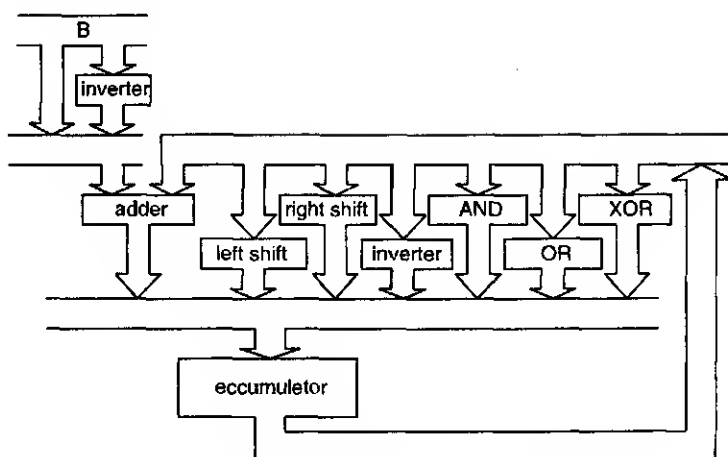
<sup>1</sup>Additional SFRs for a few other family members are shown in Appendix A4.

## Arithmetic Logic Unit (ALU)

The “heart” of the computer is the central processing unit (CPU).<sup>69</sup> With the move instructions and sequences of the processing instructions, a microcontroller can do just about anything a “big” computer can do.

<sup>68</sup>Use of the SFRs relating to interrupts and timers comes in Chapter 11 where it is immediately applicable.

<sup>69</sup>While the material that follows focuses on the way the ALU operates, remember that the CPU encompasses also the instruction fetch and interpretation hardware. It also includes the circuitry to increment the program counter and manage all the bus signals.



**The 8051 central processing unit (CPU)**

The arrows above indicate busses with tri-state outputs and latches for moving data around. The *accumulator* (ACC) always gets the output of the arithmetic/logic unit (ALU).<sup>70</sup> This is the reason why so many of the machine instructions you will see in the next chapter start with the accumulator and leave the result there.

With a series of selectable enables driven by the instruction decoder of the CPU, the accumulator can be fed the output of any of the blocks: the adder (addition—*ADD* or *ADDC*), the output of the adder with one input going through the inverter (subtraction—*SUBB* or *SUB*), one input fed only through the inverter (complement—*CPL*), or the basic logic operations (*AND*—*ANL*, *OR*—*ORL*, *XOR*—*XRL*).

Each of the logical operations involves digital gates, much like those in standard TTL design. The blocks above represent groups of eight 2-input gates working in parallel surrounded by tri-state busses to route data in and out of the blocks. Notice that this instruction leaves the result in the accumulator!<sup>71</sup>

<sup>70</sup>As best I can tell, it is called the accumulator because it *accumulates* the results of successive additions. If you feed a number in the other adder input and route the accumulator back into the other adder input, each time you latch the sum of ACC and B into ACC, the sum of B and the previous value shows up as the new ACC value.

<sup>71</sup>I would have loved to describe the same functions implemented in with ssi devices, but space restrictions and general feedback said you should learn that from a digital logic book. The parallels between discrete logic and the functions inside the CPU are exciting—you actually start to see that the CPU is not “magic.”



*ANDing* produces a high where both input bits are high. *ORing* gives a one in a place if either bit is high. *Exclusive ORing* puts out a one if one *and only one* bit is high—if both input bits are high, it puts out a zero. The inverter (*complement*) changes 0 to 1 and 1 to 0.

There are ways to move bits to the left or right in a byte, called *shifting* or *rotating*. A value of  $0001_2$  ( $1_{10}$ ) shifted left is  $0010_2$  ( $2_{10}$ ). Likewise  $1010_2$  ( $10_{10}$ ) shifted right is  $0101_2$  ( $5_{10}$ ).<sup>72</sup> The shifting is probably done directly by wiring outputs to inputs one-over from the original position. By enabling the connections to the left or right with tri-state buffers, it is possible to produce either operation, rotate left or rotate right.

Next, consider the math capabilities of the basic 8051 family.<sup>73</sup> The single-byte instructions can be put together in multi-instruction program pieces to do far more elaborate math.<sup>74</sup> Other than increment and decrement, all CPU arithmetic operations overwrite the accumulator contents with the new result.

The adder functions exactly as you would expect. The sum of the two inputs appears at the output. There are details relating to carry in and out when you get into multi-byte addition, but those are discussed with the instructions.

The key to subtracting with only an adder is the fact that, if you add one to the result of an inversion, you get the *twos complement*. Consider the number sequence of the following table:

Decimal Number	Binary Number (sign/magnitude)	Binary Number (ones complement)	Binary Number (twos complement)
3	0 00000011	00000011	00000011
2	0 00000010	00000010	00000010
1	0 00000001	00000001	00000001
0	0 00000000	00000000	00000000
-1	1 00000001	11111110	11111111
-2	1 00000010	11111101	11111110
-3	1 00000011	11111100	11111101

<sup>72</sup>There are arrangements of flip-flops called shift-registers that can do this directly. From the perspective of binary math, a shift is a multiply or divide by two.

<sup>73</sup>Again, I wanted to talk about full- and half-adders and the way the digital hardware combines to produce the math capabilities, but space limitations prevailed.

<sup>74</sup>The key to getting more elaborate math functions is the *algorithm* (ways of doing things) that can take simple math instructions of the microcontroller and do the math of statistics, trigonometry, and even calculus! At least one algorithm (several-byte addition) is discussed later in this book, but most of the coverage is in the companion book—no, I don't intend to solve differential equations on an 8051!

Inverting all the bits when a number is negative gives the ones complement, but the two's complement flips over from 0000 to 1111 the same as a hardware flip-flop chain would behave. The two's complement is the sequence you get with a counter.

Subtracting is a simple matter of inverting and adding—taking the 1's complement and feeding it into the adder (and adding 1 through the carry in) gives subtraction.<sup>75</sup> Translating that to an example, subtract two from five as follows:

---

Take the ones complement of 2	$0000010_2 \Rightarrow 11111101_2$
Add the ones complement of 2 to 5	$00000101_2 + 11111101_2 + 1 = 1\ 00000011_2$
with a 1 at the carry in <sup>1</sup>	

---

<sup>1</sup>Note that the carry out indicates that the subtraction did not *underflow*.

Subtracting 5 from 2, you have a negative answer as recognized by a lack of a carry out:

---

Take the ones complement of 5	$00000101 \Rightarrow 11111010$
Add the ones complement of 5 to 2 with a 1 at the carry in <sup>1</sup>	$11111010_2 + 00000101_2 + 1 = 0\ 11111101_2$

---

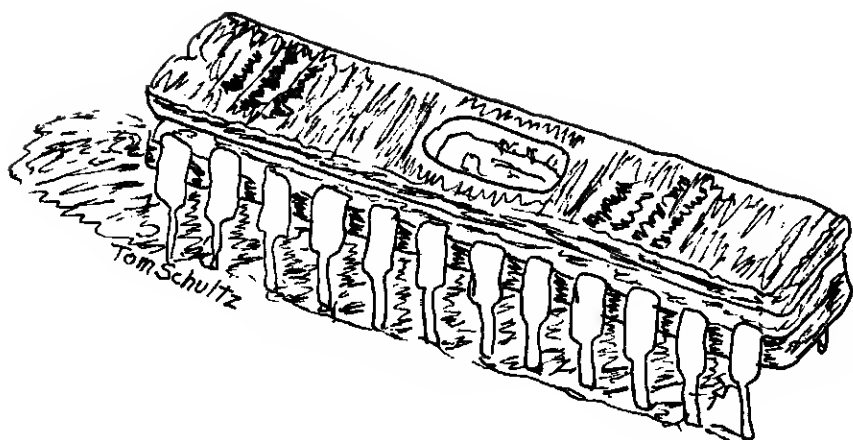
<sup>1</sup>Notice that  $11111101_2$  is the two's complement of 3. If you want the sign/magnitude representation from subtraction, invert the carry out as the sign bit and take the two's complement of the answer if it is negative.

For multibyte subtraction, you work your way from the least-significant byte up to the most-significant byte—borrows along the way are acceptable. The answer is positive (and correct) as long as there is no borrow when the most significant bytes have been subtracted. Remember that this is *unsigned* integer math. You make any adjustments for negative numbers in software that you write. *Signed* integer math is straightforward too, but I suggest you leave that for a C compiler to manage.

With the 8051 there is a hardware multiply (not shown in the diagram), which produces a 16-bit result. There is also a hardware divide. Neither operation neatly supports multibyte extensions or signed math—certainly not floating-point math! The more general approach to multiplication and division is to use algorithms made up of shifts and addition.

---

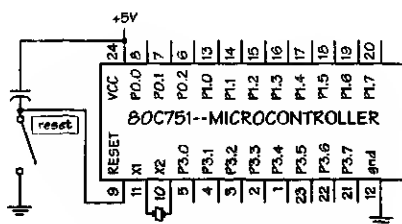
<sup>75</sup>This process is much simpler than it sounds since the number to be subtracted can be fed through simple inverters and the adding 1 is a simple matter of feeding a 1 into the carry in. A subtractor is possible directly in logic, so some processors trade complexity for speed by adding more logic and higher-level instructions to do things in fewer clock cycles. The basic principles, however, remain unchanged.



**The 87C751**

## STAND-ALONE MICROCONTROLLERS

The 8051 was probably the first single-chip computer on the market and all the versions with on-chip code (ROM, EPROM, or EEPROM) can function as a single chip. It is quite common to have the entire computer consist of the chip and a crystal, as with the 87C751.



**Smallest single-chip 8051-family microcontroller**

When you consider that the one-time programmable (OTP) version of the 87C750 costs about \$4 in small quantities, you begin to see how it can open a whole world of new applications. It can replace a few logic chips,

many of the PLDs and often a series of analog devices. Rather than struggling with a large-time-constant integrator, you can substitute a voltage to frequency converter feeding the counter on a single-chip micro. Suddenly a variety of extra features are available at no extra cost—the micro is there already and sitting relatively idle with unused ports. It can be self-calibrating, it can be adaptive, or any of a number of other possibilities to make the system more efficient, easier to calibrate, and easier to use.<sup>76</sup>

## MEMORY EXPANSION FOR THE 8051

Probably the vast majority of 8051 applications today use additional chips to hold the code and often the data as well. The expansion is straightforward—very few choices are involved if you want the result to function properly! The schematic on the next page shows a single-chip expansion of the code space which is essential for *any* application using the 8031, which has no on-chip ROM or EPROM.<sup>77</sup> The biggest drawback of memory expansion is the loss of at least two 8-bit ports to the off-chip bus functions.<sup>78</sup>

### Off-chip Code

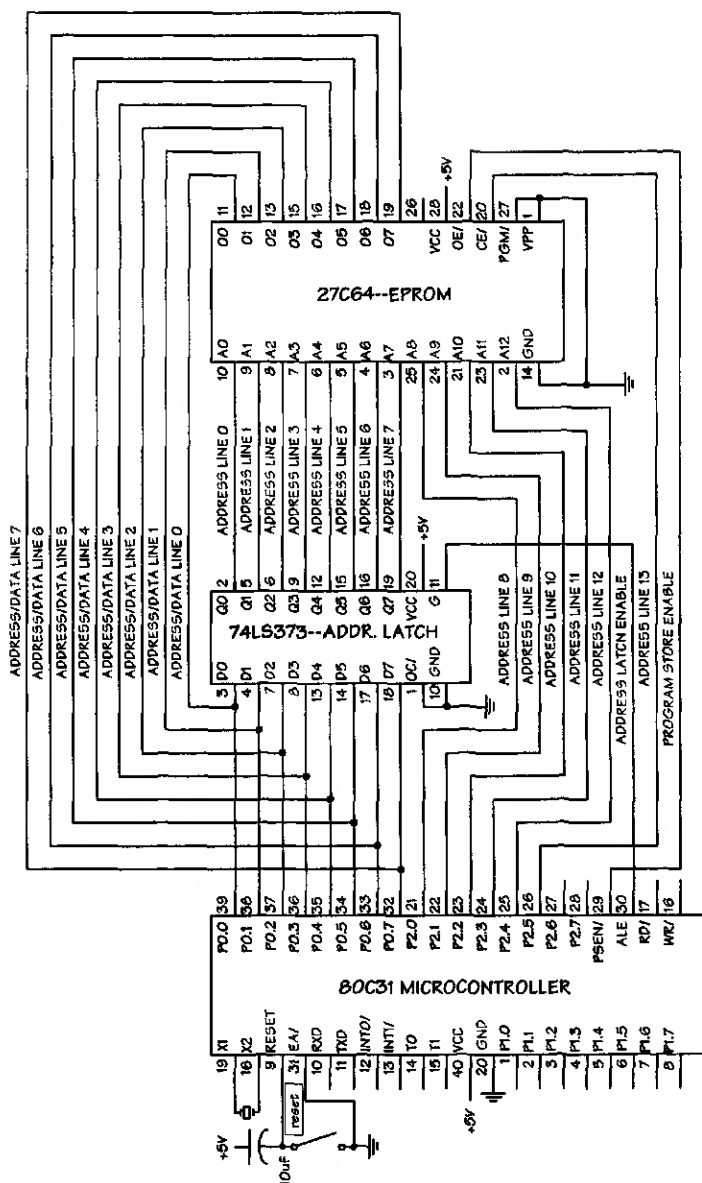
In expanding off-chip, you are limited by the multiplexed address/data information on the lower eight bits (*P0*). Look again at the timing diagrams. To hold the lower address byte until the CPU has transferred the data, you must latch it on the trailing edge of the *ALE* pulse. The 74LS373 is the common choice. The high byte of the address remains unchanging on *P2* for the entire cycle and you can use it directly for address decoding. The *PSEN*/

---

<sup>76</sup>These issues are discussed in much more detail in the companion book, *C and the 8051: Building Efficient Applications*.

<sup>77</sup>The initial purpose for on-chip EPROM versions such as the 87C51 was for prototype before committing to a factory-masked (your specific code permanently built into your specific, custom production run of 80C51 chips). Many designs use the EPROM versions with no intent of going factory-masked. But the price of the 87C51 never dropped like regular EPROMs, so unless you need the single-chip feature, use off-chip EPROM. For single-chip applications there are now much less expensive one-time programmable (OTP) versions that do not have the clear-erase window and therefore use less expensive plastic packaging. Atmel has a line of chips with EEPROM (flash memory) that are more affordable, but you still need off-chip code storage if there is not enough on-chip code space.

<sup>78</sup>You lose *P2* and *P0* (plus two bits of *P3* for *RD*/ and *WR*/ if you access off-chip RAM). There is an option of expanding only RAM using just *P0* and the two bits of *P3*. In assembly, this RAM is accessed with the *MOVX* by using *@R0* or *@R1* as the pointer. In Keil/Franklin C this memory space is called *pdata*.



8031 with only EPROM expansion—wire view

signal provides the read or output enable (*OE*) signal to the EPROM when the CPU drives the data onto the bus. In the first example, address decoding is unnecessary. With only one EPROM, the CPU never issues *PSEN*/ signal except for accessing the code on *that* chip.<sup>79</sup> The use of *A13* tied to *CS*/ is a sort of decoding since the address range where *A13* has gone high will not address the EPROM (or anything else). Thus, the code address range is from  $0000_{16}$  through  $1FFF_{16}$ .<sup>80</sup>

The second circuit shown on the next page is the same as the first but the latter schematic emphasizes the *bus* nature of the communication where the former shows every wire individually.

### Off-chip Code, Data, and Ports

Once you have given up the two ports, it is a small step to add external RAM, giving up 2 bits for *RD* and *WR*.<sup>81</sup> The next schematic shows such an expanded system with extra RAM and an 8255-port chip.<sup>82</sup> To save an address decoder, the RAM is mapped at the base ( $0000$ ) address and the 8255 is put up at the top address space. Looking at the 6164, *CS1*/ is grounded, so it is out of the picture (always enabled). *CS2* is tied to *A13*. Since it is an active-high input, *A13* must be *high* to select *this* chip. In binary, then the system has RAM from  $00100000\ 00000000_2$  to  $00111111\ 11111111_2$  ( $2000_{16}$ – $3FFF_{16}$ ). The 8255's select (*CS*/) is active-low,<sup>83</sup> so *A13* must be low to select the chip. For any address where some address bits are not connected (*don't cares*) the other bits are usually considered low, so the 8255, using only the *A0* and *A1* bits internally, would have addresses from  $00000000\ 00000000_2$  to  $00000000\ 00000011_2$  ( $2000_{16}$ – $3FFF_{16}$ ).<sup>84</sup>

<sup>79</sup>That is not to say there is *no* address decoding, but all the rest of the decoding is the circuitry *within* the EPROM that routes the reads to the specific byte within the chip in response to the 1s and 0s on the chip's address pins.

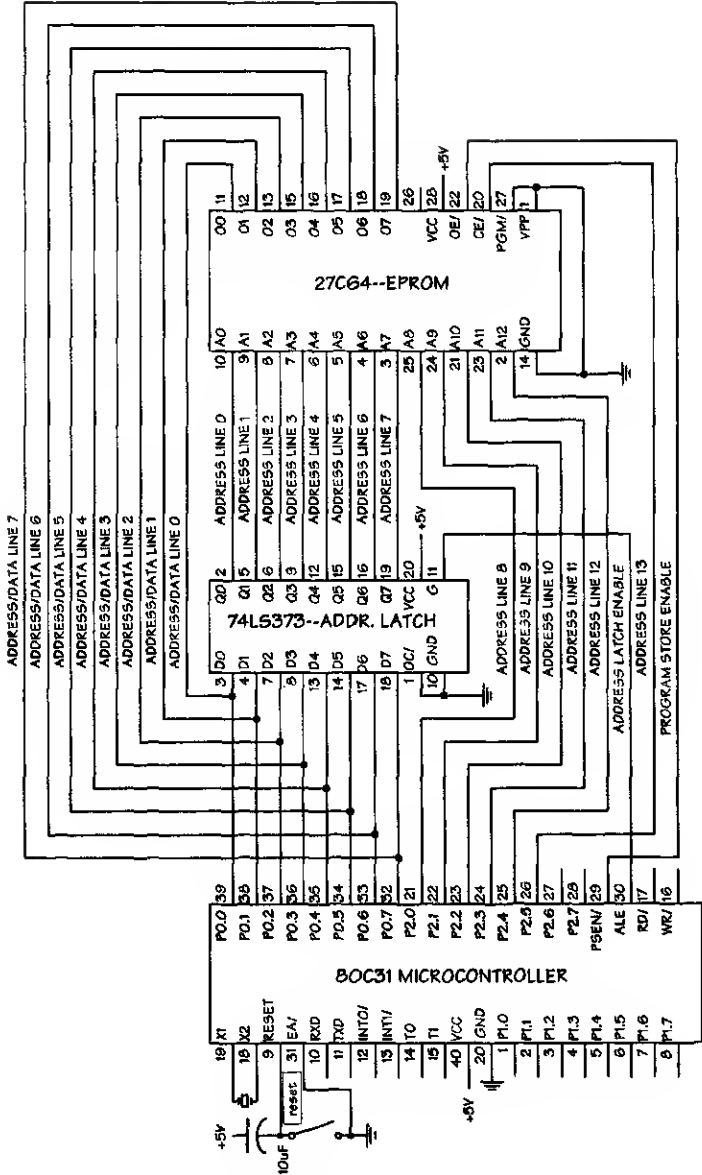
<sup>80</sup>Technically the address range would fold back since *A15* and *A14* are ignored—the code address range is then  $0000_{16}$ – $1FFF$ ,  $4000_{16}$ – $5FFF_{16}$ ,  $8000_{16}$ – $9FFF_{16}$ , and  $C000_{16}$ – $FFFF_{16}$ .

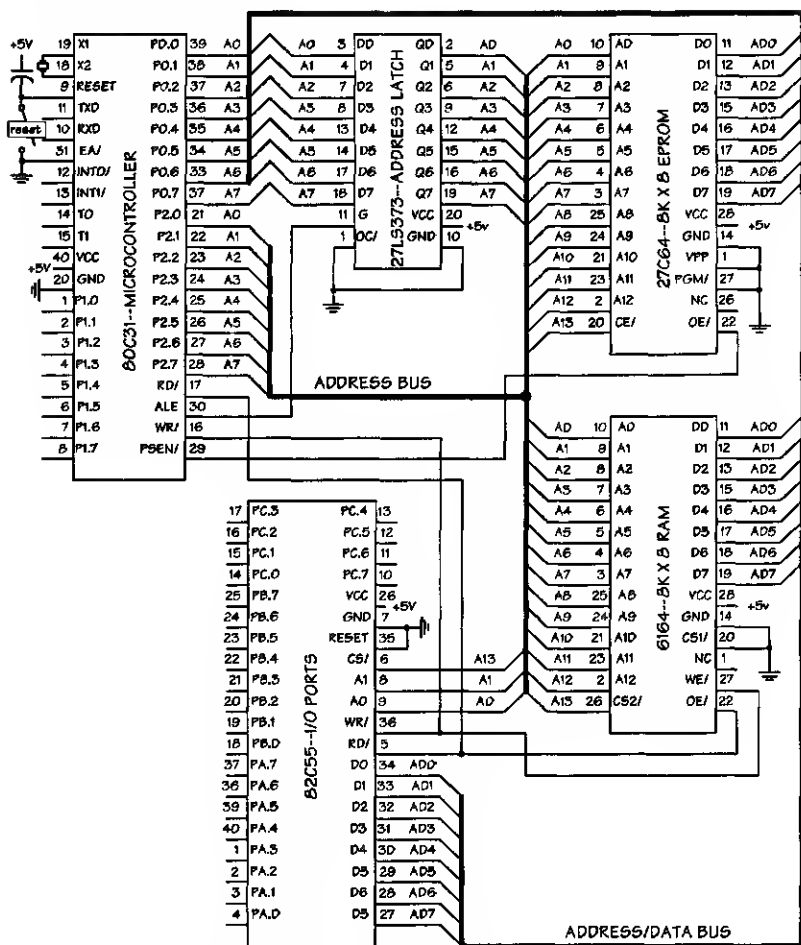
<sup>81</sup>The price of static RAM has continued to fall to the point where it is better to use generic RAM with a *separate* address latch even though Intel has the 8155 chip that provides the latch and ports along with the RAM. For three ports, it is extremely common to use the 40-pin 8255 port chip. Simple 8-bit latches for an output port and 8-bit tri-state buffers for an input port are an option as well.

<sup>82</sup>An 8255 provides three external 8-bit ports—don't worry about the internal details here.

<sup>83</sup>The term is *NOT*ed and is shown by either a bar over the top of the symbol or a / after the symbol. The bar is preferable but the / is much easier in word processors or printing.

<sup>84</sup>If you explored the details of the 8255, the four addresses for the I/O chip are: *PortA* =  $0x0000$ , *PortB* =  $0x0001$ , *PortC* =  $0x0002$ , and *CMD* =  $0x0003$ .



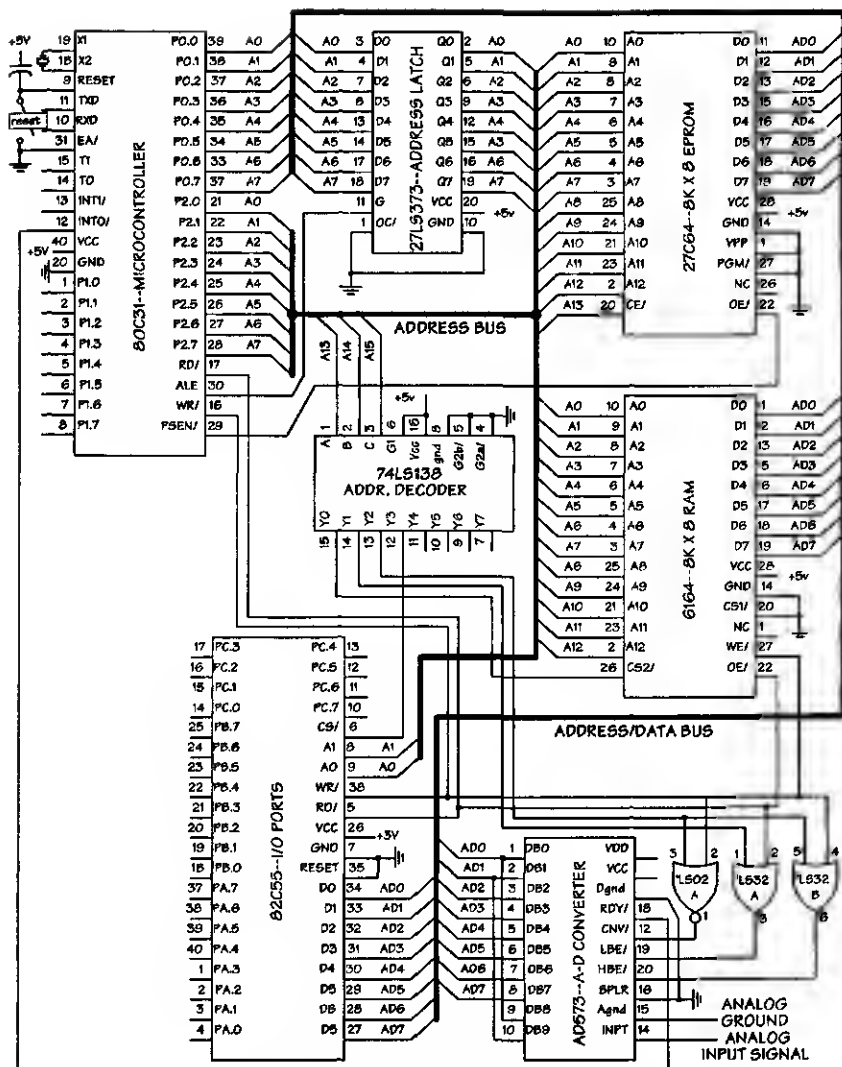


**8051 with expanded RAM, EPROM, and ports**

The RAM expansion takes up 2 bits of  $P3$  for the  $RD/$  and  $WR/$  signals. The controls for off-chip RAM ( $RD/$  and  $WR/$ ) are different from the control for EPROM ( $PSEN$ ) so code and off-chip data can be at the same numeric addresses.<sup>85</sup>

<sup>85</sup>Look at the development boards described in Appendix A4, to see that it is possible to logic OR the  $PSEN$  and  $RD$  lines so that a monitor program can download code as though it were data. The processor can then operate on that downloaded data as code. Downloading is good for quick development and debugging, but it is "cheating" since data and code spaces are supposed to be separate and could otherwise overlap in numeric addresses.





8051 with additional expansion

## Off-chip Code, Data, Ports, and A-D

Finally, on the previous page, is an example of adding another device. This particular analog to digital converter (A-D, a device that gives a binary code related to the voltage level at the input) is a 10-bit device which can be directly interfaced. Read the *RDY*/<sup>86</sup> in at one address and the *data* (the digital representation of the incoming analog voltage) at another.

Again, let us walk through the address decoding. With an 8K-byte RAM chip (needing 13 bits for internal addressing), there are only 3 bits of address for device selection, so the other 3 select lines of the 138 are permanently wired. The 8255 is enabled (a low to its *CS*/line) when the 74LS138 makes *Y3* low—011<sub>2</sub> on *CBA* (only when there is 011<sub>2</sub> on *A15–A13*). That gives addresses of 01100000 000000XX<sub>2</sub> for the 8255.<sup>87</sup> The RAM here is set at the bottom address, 00000000 00000000<sub>2</sub> to 00011111 11111111<sub>2</sub> (0000<sub>16</sub> to 1FFF<sub>16</sub>).

Interfacing the A-D takes some thought. It is a 10-bit converter so the data cannot all come back in one 8-bit read. To get the *CNV*/ pulse low requires *WR*/ low while *A15–A13* are 001<sub>2</sub>, thus, to start a conversion write *anything* to address 4000<sub>16</sub>. When the conversion is done, *RDY*/ will go low, so you can sense this bit by polling the *INT0* pin as *P3.2*—bit-address B2<sub>16</sub>.<sup>88</sup> Once the data is ready, you can read the lower 8 bits at address 4000<sub>16</sub> and the high two bits as the bottom 2 bits at address 2000<sub>16</sub>.<sup>89</sup>

After this, you can go on to expansions that are more complex. A few later examples show addition of more interrupts and memory banks. The circuitry becomes more complex, but the principles remain the same.

## Port-Driven Peripherals (LCD)

Often you will connect devices to ports rather than sacrificing the entire 16 or 18 pins needed to use the off-chip bus. On the next page is an alphanumeric LCD module tied to ports of an 87C751 (where an off-chip bus is impossible!).<sup>90</sup>

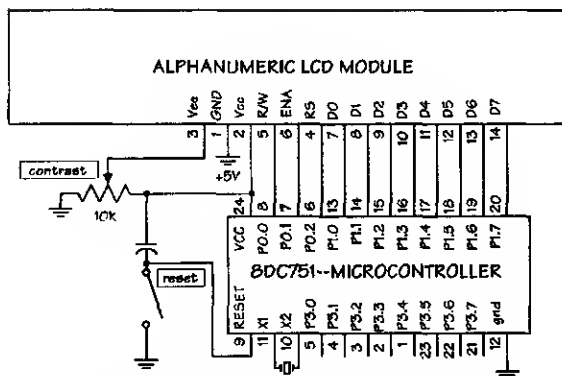
<sup>86</sup>*RDY*/ can be a signal to the microcontroller indicating when the conversion, a process that can take tens to hundreds of microsecond, is done. Writing to the *CNV*/ pin tells the A-D to begin the conversion process.

<sup>87</sup>The italic 0s are for *do not cares*, which are not decoded and could just as well be 1s. Within the 8255 the preferred addresses are: *PortA* = 0x6000, *PortB* = 0x6001, *PortC* = 0x6002, and *CMD* = 0x6003.

<sup>88</sup>You could also use an interrupt, discussed later. With the 8051-family, interrupts can be made negative-edge triggered, so this is the right polarity.

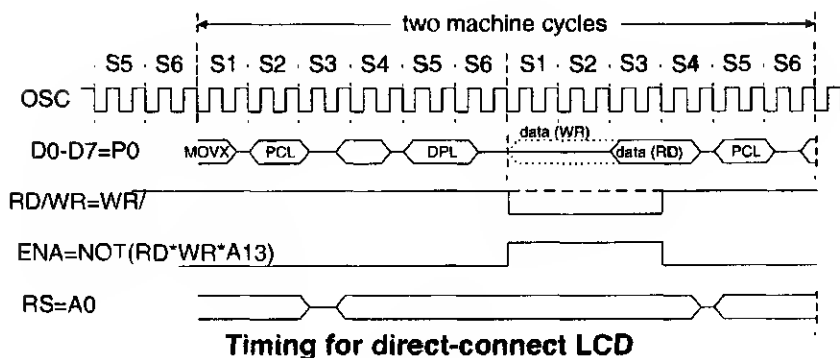
<sup>89</sup>This may not be the friendliest interface to program but, once you have figured it out, you can bury the details in a function and acquire data without a second thought!

<sup>90</sup>The software to use the LCD as shown is on page 177.



Port-driven LCD display module

### Direct connect I/O (LCD)

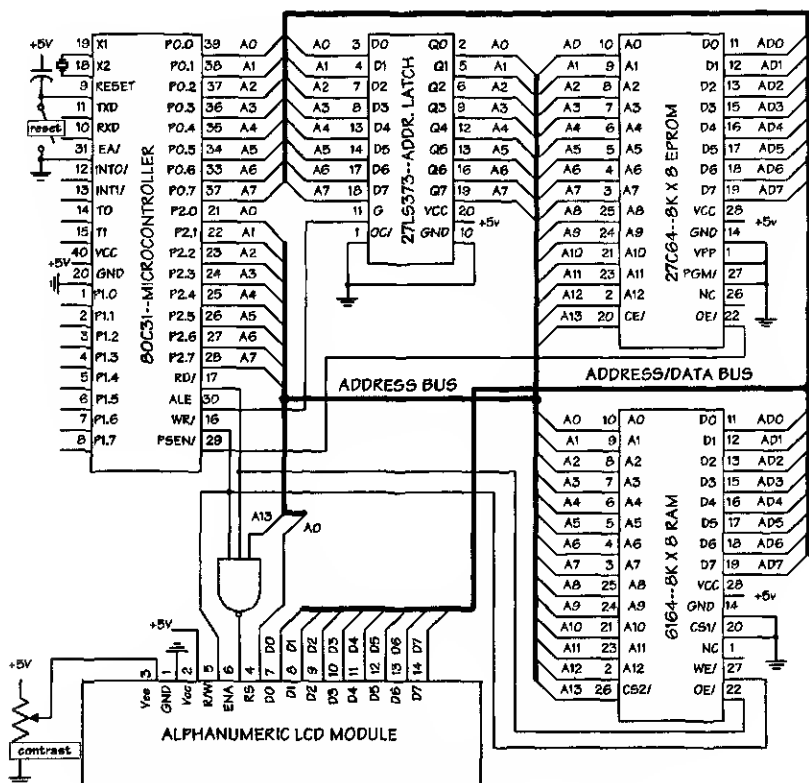


Timing for direct-connect LCD

The direct connection of the LCD to the expansion bus is a bit shaky in the schematic that follows. For starters, the duration of the enable (ENA) signal is technically too short when made up from the *RD* or *WR* signals of the 8051. It turns out that either the LCD controller chips are much faster than their specification or else there is a worst-case limit that is not reached at room temperature—in any case, the direct connection has been shown to work in numerous student projects.<sup>91</sup> The second limit is the situation where

<sup>91</sup>In a commercial application, I would be reluctant to count on such a situation without further information from the manufacturers.

the ENA goes away simultaneously with the loss of the *WR* signal. The trick is to find a source for such a delay—perhaps a string of inverters or else a latch triggered off the OSC signal to get a  $\frac{1}{2}$ -clock cycle delay. You should check the specifications for any hold time requirements. Again, I believe it actually works, but you might want to put some sort of delay in to be safe.<sup>92</sup>



## LCD module directly addressed on expansion bus

<sup>92</sup>The software to use the LCD is shown on page 177.

## REVIEW AND BEYOND

1. Give an example of a C instruction, an assembly instruction, and a machine instruction.
2. Sketch a general diagram of the architecture of a computer showing memory, I/O, the CPU, and busses.
3. What makes an 8051 an "8-bit micro"?
4. Using the techniques of this chapter, can you figure out the addresses of the PU552 and MCB520 boards in Appendix A5? Since you cannot enter the PLD in the latter, what clues can you find for the addresses?
5. In the multiaddress decoder using the 74I38, what would be the four addresses if you reversed the connections to pins 1 and 2? Would it be bad to not put EPROM at address 0000?
6. Why is dynamic RAM seldom used with microcontrollers?
7. Is a register different from a memory location? Explain.
8. What is the cardinal rule when using an 8051 internal port for input?
9. How do you convert a 1's complement number to 2's complement?
10. What is the absolute minimum you need to add to a "single-chip" microcontroller to have it work?
11. Describe two uses for the *ALE* signal.
12. With a 12 MHz crystal, how long does it take to execute the quickest 8051 instructions?
13. What are the major "costs" of expanding the RAM of the 8051?
14. Are there any setup instructions you must issue before using the two pins of *P3* as *RD/* and *WR/*?
15. With off-chip RAM attached, what happens if you write a whole byte out to port *P3*?
16. Referring to the timing diagrams on page 29, when is *P0* putting out address and when is it handling data or code?
17. Why would you possibly use the other members of the 8051 family? What features stand out in your mind?
18. Why would you use a commercial board with an 8051 rather than wiring up your own? When would you not do so? Why?

# 3

## Machine Instructions

---

If you are programming in assembly language, this chapter can help you understand the instructions.<sup>1</sup> Even if you plan to *avoid* assembly language programming, this chapter will introduce you to the instructions you will find if debugging forces you to go to a *code* listing from a compiler.

The table that follows shows the instructions all in one place for later reference. For learning the instructions, I suggest you skip over it and read about the individual categories first. The last column in the table refers to the page where you can find the detailed discussion. If you need to *disassemble* a program, see the numeric-order and alphabetic-order tables in Appendix A1. A few definitions are necessary to understand the instructions that follow:

**A:** (the accumulator or *ACC*) is the one register most heavily involved with the ALU.

**#data:** is a value preceded by a # sign is a number, not an address. It is numeric data.

**#data16:** is a 16-bit constant (2 bytes) included in the instruction.

**direct:** by itself is the designated memory location where data used by the instruction will be that found—*not* the data but an internal *ad-*

---

<sup>1</sup>The assembly language mnemonics used in this book originated with Intel (©1980), but other 8051 assemblers seem to use the same mnemonics anyway. More detailed information is available in the “MCS-51 Programmer’s Guide and Instruction Set” of Intel’s *8-bit Embedded Controllers* data book. Since Intel invented the 8051, it follows that they also invented the assembly language.

dress between 0 and  $7F_{16}$  (or  $80_{16}$  to  $FF_{16}$  for special function registers).

**Rn:** by itself refers to the *contents* of the register.

**@Ri:** preceded by an @ indicates the register is a *pointer* and refers to the value in memory where the register points. Only *R0* and *R1* can be used this way.

**DPTR:** the data pointer, used for addressing *off-chip* data and code with the *MOVX* or *MOVC* command.

**PC:** the program counter—holds the address from where the next byte of code will be fetched.

**CONFUSING A DIRECT MEMORY ADDRESS WITH DATA IN ASSEMBLY LANGUAGE!** It is painfully easy to forget the # sign and end up putting the *contents* of memory address 3 into the accumulator when you meant to fetch this *number* 3.

Mnemonic	Description	Specific Instance	Hex code	# of bytes	Clock cycle	Example pg.
----------	-------------	-------------------	----------	------------	-------------	-------------

#### DATA MOVING INSTRUCTIONS

MOV A,#data	Move immediate data to Accumulator	MOV A,#2F	E4 2F	2	12	62
MOV A,direct	Move direct byte to Accumulator	MOV A,29	E5 29	2	12	62
MOV A,@Ri	Move indirect RAM to Accumulator	MOV A,@R0 MOV A,@R1	E6 E7	1	12	62
MOV A,Rn	Move register to Accumulator	MOV A,R0 MOV A,R1 MOV A,R2 MOV A,R3 MOV A,R4 MOV A,R5 MOV A,R6 MOV A,R7	E8 E9 EA EB EC ED EE EF	1	12	62

(continued)

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
MOV direct,A	Move Accumulator to direct byte	MOV 2E,A	F5 2E	2	12	62
MOVX @Ri,A	Move ACC to External RAM (8-bit addr)	MOV @R0,A MOV @R1,A	F6 F7	1	24	62
MOV Rn,A	Move Accumulator to register	MOV R0,A MOV R1,A MOV R2,A MOV R3,A MOV R4,A MOV R5,A MOV R6,A MOV R7,A	F8 F9 FA FB FC FD FE FF	1	12	62
MOV @Ri, direct	Move direct byte to indirect RAM	MOV @R0,2E MOV @R1,2E	A6 2E A7 2E	2	24	—
MOV Rn,direct	Move direct byte to register	MOV R0,2E MOV R1,2E MOV R2,2E MOV R3,2E MOV R4,2E MOV R5,2E MOV R6,2E MOV R7,2E	A8 2E A9 2E AA 2E AB 2E AC 2E AD 2E AE 2E AF 2E	2	24	62
MOV Rn,#data	Move immediate data to register	MOV R0,#2E MOV R1,#2E MOV R2,#2E MOV R3,#2E MOV R4,#2E MOV R5,#2E MOV R6,#2E MOV R7,#2E	B8 2E B9 2E BA 2E BB 2E BC 2E BD 2E BE 2E BF 2E	2	12	63
MOV direct, direct	Move direct byte to direct	MOV 2E,29	85 2E 29	3	24	62
MOV direct, @Ri	Move indirect RAM to direct byte	MOV 2E,@R0 MOV 2E,@R1	86 2E 87 2E	2	24	—
MOV direct, Rn	Move register to direct byte	MOV 2E,R0 MOV 2E,R1 MOV 2E,R2 MOV 2E,R3 MOV 2E,R4	88 2E 89 2E 8A 2E 8B 2E 8C 2E	2	24	62

(continued)



Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
		MOV 2E,R5	8D 2E			
		MOV 2E,R6	8E 2E			
		MOV 2E,R7	8F 2E			
MOV direct, #data	Move immediate data to direct byte	MOV 2E,#2F	75 2E 2F	3	24	63
MOV @Ri, #data	Move immediate data to indirect RAM	MOV @R0,#2E MOV @R1,#2E	76 2E 77 2E	2	12	63
MOV DPTR, #data16	Load Data Pointer with a 16-bit constant	MOV DPTR, 21F5H	90 21 F5	3	24	63
MOVC A, @A+DPTR	Move Code byte relative to DPTR to ACC	MOVC A, @A+DPTR	93	1	24	63
MOVC A, @A+PC	Move Code byte relative to Pgm Cntr to ACC	MOVC A, @A+PC	83	1	24	63
MOVX A, @DPTR	Move External RAM (16-bit addr) to ACC	MOVX A, @DPTR	E0	1	24	64
MOVX @DPTR, A	Move ACC to External RAM (16-bit addr)	MOVX @DPTR, A	F0	1	24	64

#### EXCHANGE (BYTE SWAP) OPERATIONS

XCH A, direct	Exchange direct byte with Accumulator	XCH A,2E	C5 2E	2	12	64
XCH A, @Ri	Exchange indirect RAM with Accumulator	XCH A,@R0 XCH A,@R1	C6 C7	1	12	64
XCH A, Rn	Exchange register with Accumulator	XCH A,R0 XCH A,R1 XCH A,R2 XCH A,R3 XCH A,R4 XCH A,R5 XCH A,R6 XCH A,R7	C8 C9 CA CB CC CD CE CF	1	12	64

#### STACK OPERATIONS

PUSH direct	Push direct byte onto stack	PUSH 2E	C0 2E	2	24	65
POP direct	Pop direct byte from stack	POP 2E	D0 2E	2	24	66

(continued)

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
----------	-------------	-------------------	----------	------------	-------------	-------------

### UNCONDITIONAL BRANCHING (JUMP) INSTRUCTIONS

AJMP addr 11	Absolute Jump	AJMP 06F3	C1 F3 A10A9 A8 0 0001	2	24	66
LJMP addr 16	Long Jump	LJMP 21E7	02 21 E7	3	24	66
SJMP rel	Short Jump (relative addr)	SJMP E5	80 E5	2	24	66
JMP @A+DPTR	Jump indirect relative to the DPTR	JMP @A+DPTR	73	1	24	67

### CONDITIONAL BRANCHING INSTRUCTIONS

JZ rel	Jump if Accumulator is zero	JZ E5	60 E5	2	24	67
JNZ rel	Jump if Accumulator is not zero	JNZ E5	70 E5	2	24	67
CJNE A, direct, rel	Compare direct byte to ACC and jump if not equal	CJNE A,2E,E5	B5 2E E5	3	24	67
CJNE A, #data, rel	Compare immediate to ACC and jump if not equal	CJNE A,#19,E5	B4 19 E5	3	24	67
CJNE Rn, #data, rel	Compare immediate to register and jump if not equal	CJNE R0,#19,E5 CJNE R1,#19,E5 CJNE R2,#19,E5 CJNE R3,#19,E5 CJNE R4,#19,E5 CJNE R5,#19,E5 CJNE R6,#19,E5 CJNE R7,#19,E5	D8 19 E5 D9 19 E5 DA 19 E5 DB 19 E5 DC 19 E5 DD 19 E5 DE 19 E5 DF 19 E5	3	24	67
CJNE @Ri, #data, rel	Compare immediate to indirect and jump if not equal	CJNE @R0,#19,E5 CJNE @R1,#19,E5	B6 19 E5 B7 19 E5	3	24	67
DJNZ Rn, rel	Decrement register and jump if not zero	DJNZ R0,2E DJNZ R1,2E DJNZ R2,2E DJNZ R3,2E DJNZ R4,2E DJNZ R5,2E DJNZ R6,2E DJNZ R7,2E	D8 2E D9 2E DA 2E DB 2E DC 2E DD 2E DE 2E DF 2E	2	24	68

(continued)

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
DJNZ A, direct, rel	Decrement direct byte and jump if not zero	DJNZ A,2E,E5	D5 2E E5	3	24	68
JC rel	Jump if Carry is set	JC F5	40 F5	2	24	67
JNC rel	Jump if Carry not set	JNC F5	50 F5	2	24	67
JB bit, rel	Jump if direct Bit is set	JR 93,F5	20 93 F5	3	24	67
JNB bit, rel	Jump if direct Bit is not set	JNB 93,F5	30 93 F5	3	24	67
JBC bit, rel	Jump if direct Bit is set & clear bit	JBC 93,F5	10 93 F5	3	24	67

#### SUBROUTINE CALL INSTRUCTIONS

ACALL addr11	Absolute Subroutine Call	ACALL 06F3	D1 F3 A10A9A8 10001	2	24	69
LCALL addr16	Long Subroutine Call	LCALL 21E7	12 21 E7	3	24	69
RET	Return from Subroutine	RET	22	1	24	70
RETI	Return from interrupt	RETI	32	1	24	70

#### NO OPERATION INSTRUCTION

NOP	No Operation	NOP	0	1	12	70
-----	--------------	-----	---	---	----	----

#### LOGICAL AND OPERATIONS

ANL direct, A	AND Accumulator to direct byte	ANL 2E,A	52 2E	2	12	70
ANL direct, #data	AND immediate data to direct byte	ANL 2E,#19	53 2E 19	3	24	—
ANL A, #data	AND immediate data to Accumulator	ANL A,#2F	54 2F	2	12	71
ANL A, direct	AND direct byte to Accumulator	ANL A,29	55 29	2	12	70
ANL A, @Ri	AND indirect RAM to Accumulator	ANL A,@R0	56	1	12	70
		ANL A,@R1	57			
ANL A, Rn	AND Register to Accumulator	ANL A,R0	58	1	12	70
		ANL A,R1	59			
		ANL A,R2	5A			
		ANL A,R3	5B			

(continued)

Mnemonic	Description	Specific Instance	Hex code	# of bytes	Clock cycle	Example pg.
		ANL A,R4	5C			
		ANL A,R5	5D			
		ANL A,R6	5E			
		ANL A,R7	5F			

**LOGIC OR OPERATIONS**

ORL direct, A	OR Accumulator to direct byte	ORL 2E,A	42 2E	2	12	71
ORL direct, #data	OR immediate data to direct byte	ORL 2E,#19	43 2E 19	3	24	—
ORL A, #data	OR immediate data to Accumulator	ORL A,#2F	44 2F	2	12	71
ORL A, direct	OR direct byte to Accumulator	ORL A,29	45 29	2	12	71
ORL A, @Ri	OR indirect RAM to Accumulator	ORL A,@R0 ORL A,@R1	46 47	1	12	71
ORL A, Rn	OR register to Accumulator	ORL A,R0 ORL A,R1 ORL A,R2 ORL A,R3 ORL A,R4 ORL A,R5 ORL A,R6 ORL A,R7	48 49 4A 4B 4C 4D 4E 4F	1	12	71

**EXCLUSIVE OR OPERATIONS**

XRL direct, A	Exclusive OR Accumulator to direct byte	XRL 2E,A	62 2E	2	12	71
XRL direct, #data	Exclusive OR immediate data to direct byte	XRL 2E,#19	63 2E 19	3	24	—
XRL A, #data	Exclusive OR immediate data to Accumulator	XRL A,#2F	64 2F	2	12	71
XRL A, direct	Exclusive OR direct byte to Accumulator	XRL A,29	65 29	2	12	71
XRL A, @Ri	Exclusive OR indirect RAM to Accumulator	XRL A,@R0 XRL A,@R1	66 67	1	12	71
XRL A, Rn	Exclusive OR register to Accumulator	XRL A,R0 XRL A,R1	68 69	1	12	71

(continued)

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
		XRL A,R2	6A			
		XRL A,R3	6B			
		XRL A,R6	6C			
		XRL A,R5	6D			
		XRL A,R6	6E			
		XRL A,R7	6F			

#### CLEAR AND COMPLEMENT OPERATIONS

CPL A	Complement Accumulator	CPL A	F4	1	12	71
CLR A	Clear Accumulator	CLR A	E4	1	12	71

#### ROTATE OPERATIONS

RR A	Rotate Accumulator Right	RR A	3	1	12	72
RRC A	Rotate Accumulator Right through the Carry	RRC A	13	1	12	72
RL A	Rotate Accumulator Left	RL A	23	1	12	72
RLC A	Rotate Accumulator Left through the Carry	RLC A	33	1	12	72

#### BOOLEAN VARIABLE MANIPULATION

CLR C	Clear Carry	CLR C	C3	1	12	74
CLR bit	Clear direct bit	CLR 93	C2 93	2	12	74
SETB C	Set Carry	SETB C	D3	1	12	74
SETB bit	Set direct bit	SETB 93	D2 93	2	12	74
CPL C	Complement Carry	CPL C	B3	1	12	74
CPL bit	Complement direct bit	CPL 93	B2 93	2	12	74
ANL C, bit	AND direct bit to Carry	ANL C,93	82 93	2	24	74
ANL C, /bit	AND complement of direct bit to Carry	ANL C,/93	B0 93	2	24	74
ORL C, bit	OR direct bit to Carry	ORL C,93	72 93	2	24	74
ORL C, /bit	OR complement of direct bit to Carry	ORL C,/93	A0 93	2	24	74
MOV C, bit	Move direct bit to Carry	MOV C,93	A2 93	2	12	75
MOV bit, C	Move Carry to direct bit	MOV 93,C	92 93	2	24	75

(continued)

Mnemonic	Description	Specific Instance	Hex code	# of bytes	Clock cycle	Example pg.
----------	-------------	-------------------	----------	------------	-------------	-------------

**ADDITION OPERATIONS**

ADD A, #data	Add immediate data to Accumulator	ADD A,#2F	24 2F	2	12	75
ADD A, direct	Add direct byte to Accumulator	ADD A,29	25 29	2	12	75
ADD A, @Ri	Add indirect RAM to Accumulator	ADD A,@R0 ADD A,@R1	26 27	1	12	75
ADD A, Rn	Add register to Accumulator	ADD A,R0 ADD A,R1 ADD A,R2 ADD A,R3 ADD A,R4 ADD A,R5 ADD A,R6 ADD A,R7	28 29 2A 2B 2C 2D 2E 2F	1	12	75
ADDC A, #data	Add immediate data to ACC with Carry	ADDC A,#2F	34 2F	2	12	76
ADDC A, direct	Add direct byte to Accumulator with Carry	ADDC A,29	35 29	2	12	75
ADDC A, @Ri	Add indirect RAM to Accumulator with Carry	ADDC A,@R0 ADDC A,@R1	36 37	1	12	76
ADDC A, Rn	Add register to Accumulator with Carry	ADDC A,R0 ADDC A,R1 ADDC A,R2 ADDC A,R3 ADDC A,R4 ADDC A,R5 ADDC A,R6 ADDC A,R7	38 39 3A 3B 3C 3D 3E 3F	1	12	75

**SUBTRACT OPERATIONS**

SUBB A, #data	Subtract immediate data from ACC with borrow	SUBB A,#2F	94 2F	2	12	76
SUBB A, direct	Subtract direct byte from ACC with borrow	SUBB A,29	95 29	2	12	76
SUBB A, @Ri	Subtract indirect RAM from ACC with borrow	SUBB A,@R0 SUBB A,@R1	96 97	1	12	76

(continued)

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
SUBB A, Rn	Subtract Register from ACC with borrow	SUBB A,R0	98	1	12	76
		SUBB A,R1	99			
		SUBB A,R2	9A			
		SUBB A,R3	9B			
		SUBB A,R4	9C			
		SUBB A,R5	9D			
		SUBB A,R6	9E			
		SUBB A,R7	9F			

#### INCREMENT AND DECREMENT OPERATIONS

INC A	Increment Accumulator	INC A	04	1	12	77
INC direct	Increment direct byte	INC 2E	05 2E	2	12	77
INC @Ri	Increment indirect RAM	INC @R0	06	1	12	77
		INC @R1	07			
INC Rn	Increment register	INC R0	08	1	12	77
		INC R1	09			
		INC R2	0A			
		INC R3	0B			
		INC R4	0C			
		INC R5	0D			
		INC R6	0E			
		INC R7	0F			
DEC A	Decrement Accumulator	DEC A	14	1	12	77
DEC direct	Decrement direct byte	DEC 2E	15 2E	2	12	77
DEC @Ri	Decrement indirect RAM	DEC @R0	16	1	12	77
		DEC @R1	17			
DEC Rn	Decrement Register	DEC R0	18	1	12	77
		DEC R1	19			
		DEC R2	1A			
		DEC R3	1B			
		DEC R4	1C			
		DEC R5	1D			
		DEC R6	1E			
		DEC R7	1F			
INC DPTR	Increment Data Pointer	INC DPTR	A3	1	24	77

#### MULTIPLY AND DIVIDE OPERATIONS

MUL AB	Multiply A and B	MUL AB	A4	1	48	77
DIV AB	Divide A by B	DIV AB	84	1	48	77

Mnemonic	Description	Specific instance	Hex code	# of bytes	Clock cycle	Example pg.
<b>DECIMAL MATH OPERATIONS</b>						
XCHD A,@Ri	Exchange low-order digit indirect RAM with ACC	XCHD A,@R0 XCHD A,@R1	D6 D7	1	12	77
SWAP A	Swap nibbles within the Accumulator	SWAP A	C4	1	12	78
DA A	Decimal Adjust Accumulator	DA A	D4	1	12	78

## DATA MOVING INSTRUCTIONS

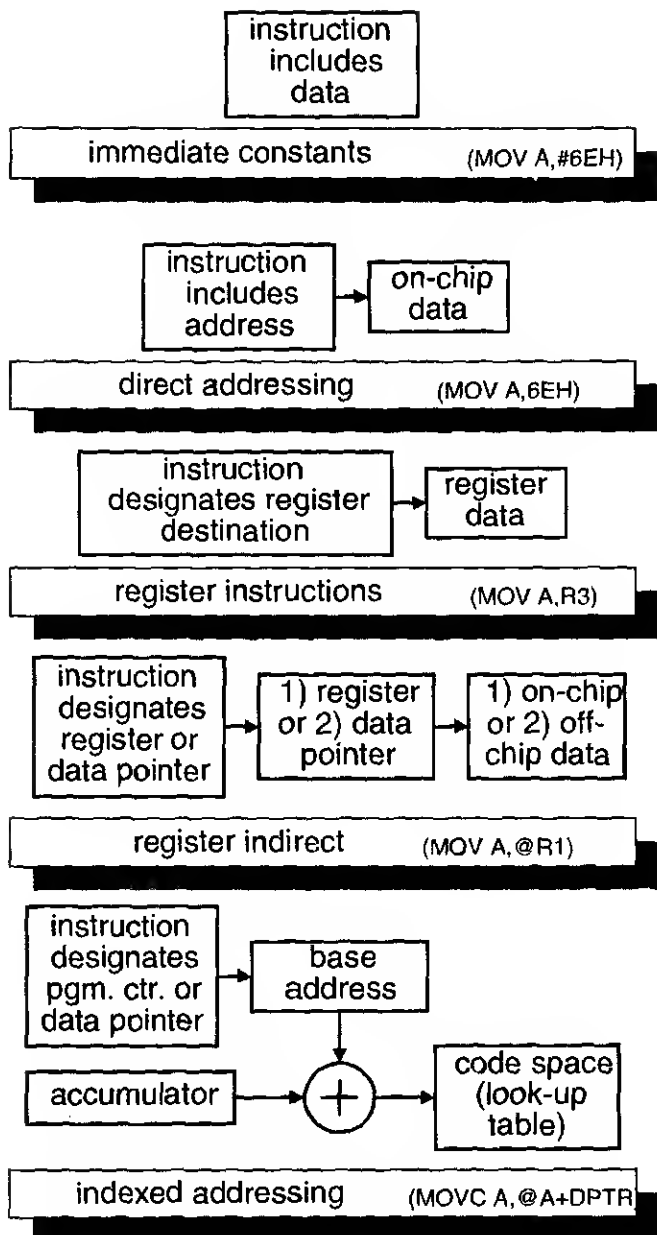
The first instructions are the ones to move numbers around between the registers and the various types of memory.

There are several ways to do this, sometimes called *addressing modes*, shown in the figure on the next page.<sup>2</sup>

<sup>2</sup>All of the lower 128 bytes of RAM (or 64 for those smaller ones), up to address  $7F_{16}$ , are **directly addressable**. That means that there are instructions to get to this data that include the address right in the instruction. The opposite, **indirectly addressable**, means you first have to issue an instruction to set up a pointer value and then issue a second instruction to actually move the data found there. An example of a direct addressing instruction is *MOV ACC, @043H* where the instruction moves the *contents* of internal address  $43_{16}$  into a register in the CPU called the ACCumulator.

Indirect addressing instructions for internal RAM must first set *R0* or *R1*. These instructions can access both the lower, directly addressable RAM ( $00$  to  $7F_{16}$ ) and the indirectly addressable RAM (from  $80_{16}$  to  $FF_{16}$ ). For example, to fetch the contents of address  $C4_{16}$  you would need two instructions, *MOV R1, 0C4H* and then *MOV ACC, @R1* to first set up the *R1 pointer* and then get the data from where the pointer pointed. The only way to get to the upper bytes (addresses  $80_{16}$  to  $FF_{16}$ ) is through *R0* or *R1* (Technically, I should add that the *stack* can be set up here, but that will be covered later).





Addressing modes for the 8051-family

The move instruction has many different forms depending on where the data comes from and where it goes. *MOV* does not destroy the data in the source—rather it copies it to the destination. A few combinations that you might expect do not exist as instructions. For example, you cannot move data from one register to another, but once you realize that, for a given register bank, each register also has a direct address, you can get at them directly.<sup>3</sup> There is also no indirect/indirect move. Two-bit move instructions are also grouped with the Boolean ones.

**SPECIAL FUNCTION REGISTERS** are usually referred to by name (already known to the assembler) so you can write  
`MOV TCON, #013H`

**SUBSTITUTE NAMES FOR NUMBERS:** In assembly use equates in place of the direct addresses so you have  
 at the top: `RELAYSTATUS EQU 045H`  
 and down below: `MOV RELAYSTATUS, #0FFH`

Notice the significance of the # and @ symbols. It is very common to forget the # when intending to move data—the result is a fetch from some direct address, which may be holding anything!

Some instructions directly access the bottom 32 bytes of internal RAM as four *registerbanks* (referred to as *bank 0* through *bank 3*) each with eight registers (referred to as *R0* through *R7*). For example, there is an instruction to move the contents of the accumulator into *R7*, `MOV R7, ACC`. The earlier set of instructions can still directly address the *same* memory space as addresses 0–7, 8–F, 10–17 and 18–1F. When referring to a register (say *R3*), the actual address (03, 0B, 13, or 1B) depends on which bank is in use. Any unused registerbanks as well as the rest of the 128 (or more) bytes of internal memory can be available for direct use.

The locations from byte addresses 20<sub>16</sub> through 2F<sub>16</sub> (16 bytes—128 bits) are also usable as *bit-addressable* memory. Single bits from bit addresses 0 to 7F<sub>16</sub> can be set or cleared, as with `SETB 03EH` which would put a 1 in bit location 3E<sub>16</sub> (bit 6 of byte address 27<sub>16</sub> or 27.6). This RAM is the same and you can access it as bits or as bytes (with different instructions).

<sup>3</sup>With the Keil/Franklin C compiler, you can prohibit this addressing—for a function independent of the current register bank. See the parameter settings in Chapter 9. If a program is using *bank0*, a reference to *R7* would be to address 07<sub>16</sub> whereas in *bank1* a reference would be to address 0F<sub>16</sub>. As long as the instruction only used *R7*, the reference would be correct for either.

### Accumulator/Register

---

MOV A, R7 ; copies the *contents* of R7 to the accumulator  
 MOV R1, A ; copies the *contents* of the accumulator to R7

---

### Accumulator/Direct

---

MOV A, 22H ; copies *contents* of location 22<sub>16</sub> to accumulator  
 MOV 03, A ; accumulator  $\Rightarrow$  3<sub>10</sub> (R3 in registerbank 0)  
 MOV A, 1BH ; 1B<sub>16</sub> (R3 in registerbank 3)  $\Rightarrow$  accumulator

---

### Accumulator/Data

---

MOV A, #22 ; the *number* 22<sub>10</sub> (16<sub>16</sub>)  $\Rightarrow$  accumulator  
 MOV A, #7EH ; the *number* 7E<sub>16</sub>  $\Rightarrow$  accumulator

---

### Register/Data

---

MOV R1, #5FH ; the *number* 5F<sub>16</sub>  $\Rightarrow$  R1

---

### Accumulator/Indirect

This can only be done with R0 or R1 as the pointer.

---

MOV R1, #4BH ; the *number* 4B<sub>16</sub>  $\Rightarrow$  R1 (setup for next lines)  
 MOV A, @R1 ; *contents of where R1 points* (4B<sub>16</sub>)  $\Rightarrow$  accumulator  
 MOV @R0, A ; accumulator  $\Rightarrow$  *where R0 points*

---

### Register/Direct

---

MOV R3, 7FH ; *contents* of location 7F<sub>16</sub>  $\Rightarrow$  R3  
 MOV 6EH, R2 ; R2  $\Rightarrow$  location 6E<sub>16</sub>

---

### Direct/Direct

---

MOV 1FH, 7EH ; *contents* of location 7E<sub>16</sub>  $\Rightarrow$  *location* 1F<sub>16</sub>

---

### Direct/Indirect

---

MOV R3, @R1 ; *contents* of location *pointed to by* R1  $\Rightarrow$  R3  
 MOV @R0, R6 ; R6  $\Rightarrow$  *location pointed to by* R0

---

**Direct/Data**


---

MOV R7, #01 ; the number 01<sub>10</sub> ⇒ R7

---

**Indirect/Data**


---

MOV @R0, #7FH ; the number 7f<sub>16</sub> ⇒ where R0 points

---

**Data Pointer/Data**

This is a 2-byte load instruction that takes the 16-bit number and puts it into the pointer (*DPTR*) that is used for accessing off-chip memory (by *MOVX*).

---

MOV DPTR, #0FFC0H ; the number C0<sub>16</sub> ⇒ DPL; the number FF<sub>16</sub> ⇒ DPH

---

**MOVC**

Access to code space (usually EPROM) is, by definition, read-only.<sup>4</sup> The *MOVC* is quite useful for fetching data from ROM-based lookup tables. It is the only instruction to use the indexed addressing mode.

---

MOVC A, @A+DPTR ; contents of off-chip code location (ROM) pointed to by sum of DPTR and ACC ⇒ ACC (note these destroy the value in ACC)

MOVC A, @A+PC ; contents of code location ACC-bytes down (higher address) from code pointed to by current program counter ⇒ ACC

---

I have never seen the latter instruction actually used, but it *could* be used to fetch values from a table stored just following the instructions, as shown in detailed instruction discussion in the Intel data book. I have no clue how you could force a C compiler to duplicate this clever trick if it didn't decide to do it on its own. The linker would probably fight linking constants in the code segment.

---

<sup>4</sup>Some systems will OR the *PSEN* and *RD* lines to produce overlapping addresses in RAM so downloading of code from a host development system is possible, but the rule is still read-only for code. See Appendix A4 on the development boards for a more detailed discussion of this combining of code and data space for downloading.

## MOVX

This is the mainstay of most large programs because no instructions work directly on external RAM and most applications have become too large to work with less than 256 bytes of storage.<sup>5</sup>

---

```
MOV DPTR, #021C5H ; 2116 ⇒ DPH; C516 ⇒ DPL (set up for next
lines)
MOVX A, @DPTR off-chip RAM contents at location pointed to by DPTR
(here 21C516) ⇒ ACC
MOVX @DPTR, A ; ACC ⇒ location pointed to by DPTR A (here 21C516)
MOVX A, @R0 ; 8-bit off-chip location's contents ⇒ ACC
MOVX @R1, A ; ACC ⇒ 8-bit off-chip location
```

---

These are the standard instructions for addressing external I/O ports as well as other variables. With C this two-instruction process is transparent to the programmer.

## XCH

Unlike the *MOV* that copies from one place to another, the *XCH* swaps the 2 bytes. It is particularly useful with the many operations that must go through the accumulator.

### Accumulator/Register

---

```
XCH A, R4 ; contents of R4 and ACC are swapped
```

---

### Accumulator/Direct

---

```
XCH A, 1EH ; contents of location 1E16 and ACC are swapped
```

---

### Accumulator/Indirect

---

```
MOV R1, #05BH ; put 5B16 into R1's location (setup for next line)
XCH A, @R1 ; contents of location pointed to by R1 (here the internal
RAM address 5B16) and ACC are swapped
```

---

<sup>5</sup>The *page MOVX* using *R0* or *R1* is quite seldom used; it only puts out the 8-bit address on *P0* and doesn't affect *P2*. If you should need lots of ports and no external RAM, it could handle up to 256 external ports. Alternately, you could put out a page select on *P2* just before using this instruction. These approaches are for systems where lack of a few port pins would destroy the cost savings of a minimal system and are not used much.

## STACK INSTRUCTIONS

There is another way of storing data—the *stack*. Up to now you have put a byte of information at a specific location—for example, the reading that comes in from the switches. Then you added three to it and put the result back in the same place—one *specific place*. With a **stack**, values are stored in successive locations addressed by the **stack pointer**. Each time you *push* one value onto the stack, the stack pointer increments by one. Each time you *pop* a value from the stack, the stack pointer decrements by one. It is a first-in last-out buffer.

With the 8051 family, the stack is in on-chip RAM.<sup>6</sup> Two instructions put bytes on the stack. First, the *PUSH* instruction puts a byte on every time it is used. Second, as is discussed later, every *CALL* instruction and every hardware interrupt put the current program counter values (2 bytes) onto the stack.<sup>7</sup>

### PUSH

This instruction puts a byte onto the stack. The contents of any of the direct addresses can be pushed including the SFRs. Note that the stack pointer points to the top value—not the empty space above the top. It is possible to push *ACC*, *B*, *PSW*, and the various hardware control registers. It is not possible to push *R0* through *R7* by name because you are expected to switch registerbanks by changing two bits of *PSW* (discussed in Chapter 2).<sup>8</sup>

---

```
MOV SP, #09CH    set stack pointer to 9C16
PUSH B           increment the stack pointer to 9D16 and put contents of register B
                  (found at direct address F016) onto stack at internal memory location 9D16
```

---

<sup>6</sup>The stack is wherever the startup program puts it by setting the stack pointer value. In C the compiler and linker handle the details, but in assembly you have to be careful to leave enough empty RAM above the stack so the stack will not *overflow*.

<sup>7</sup>This is so the return (*RET* or *RETI*) can restore the program counter to where it left off when the subroutine ends. I discuss subroutines and functions more with the *CALL* instructions and later under Functions (Chapter 7).

<sup>8</sup>This gives a way to change the program counter artificially by pushing it, altering the stack or stack pointer, and popping back to a new address. It is a key to context switching in multi-tasking operating systems.

## POP

This is the reverse of the *PUSH*. Remember when restoring after multiple *PUSH*es that the sequence should be in reverse order. Out of order, push/pop can be an easy way to swap values.

---

POP PSW ; (assuming SP is at  $9C_{16}$  and there is a value of  $38_{16}$  at internal RAM location  $9C_{16}$ ) top of stack (here  $38_{16} \Rightarrow$  PSW (direct address  $D0_{16}$ ) and SP is decremented to  $9B_{16}$ )

---

## BRANCHING INSTRUCTIONS

Branching instructions re-route the program execution sequence. There are actually three different addressing methods for the jump and call instructions.<sup>9</sup> The *short* jump (*SJMP*) covers an address range from 128 bytes back to 127 bytes forward from the address of the next instruction. The *absolute* branches, *AJMP* and *ACALL* supply the lower 11 bits of the 16-bit address and keep the upper 5 bits of the next program instruction. This forces the destination to be in the same 2K block as the *CALL*. It makes a nightmare for the linker designer (or the assembly programmer) when relocatable code is involved, but it uses only two bytes of code rather than three. Finally, there are the *long* branches, *LCALL* and *LJMP* that include the full, absolute 16-bit address of the destination.<sup>10</sup>

### Unconditional JMP

These are jumps that occur without testing. I show them with *label* locations since that is the way to write *readable* assembly code, but you *could* put in a specific numeric code address with a # (as *#215EH*). The instruction causes the program counter contents (*PC* register) to change from pointing to the next instruction after the jump instruction, to instead pointing to the new address. The next instruction to execute is then the one at the new address.

---

AJMP SUB in same 2k block  
 LJMP POINTA anywhere in code  
 SJMP LOOP relative: +127 to -128

---

<sup>9</sup>Many assemblers will accept *JMP* and *CALL* as the instruction and determine the most efficient instruction for you. The C compiler handles the choice for you.

<sup>10</sup>Incidentally, it is the fact that all but the *SJMP* branch have *absolute* addresses that makes it virtually impossible to have an 8051 operating system that swaps in and out relocatable programs as is commonly done on a PC.

JMP @A+DPTR this instruction could be used to do a multidirection branching (a switch case in C), but I don't think it is used much. Even with a jump table the value for ACC would have to be multiplied by two or three to fit the jump instructions.

### Conditional JMP

These test and either make a *short* jump or else flow on to the next instruction.

```

JZ  POINTX    ; if ACC is all zero
JNZ  POINTY    ; if any bits of ACC are not zero
JC   POINTZ    ; jump if carry flag is 1 (set)
JNC  POINTZ
JB   P3.5, POINTA    ; test specific bit (here in port 3)
JNB  06EH, POINTB    ; test bit in bit-addressable RAM area
JBC  22.3, POINTC    ; this also clears the tested bit (here 22.3 = bit 1910
                     = bit 1316)

```

## Comparisons

Comparing is, to the ALU, a matter of subtracting (adding the inverse) and checking for a carry. There are comparison ICs that report three results on the relative magnitude of two binary numbers (*equal*, *greater than*, or *less than*), but the 8051 ALU avoids the extra gates by using the carry from subtraction.

**CJNE**

*CJNE* is *compare and jump* (short) if *not equal*. Not all combinations exist. You can only compare a register with the accumulator using the register's *direct address*, which depends on the registerbank. You cannot, for example, refer to *R0* or *R7*, but if you are using registerbank 0, you can refer to 00 or 07 (or 08 and 0FH if you are using registerbank 1).

```
CJNE A, 3EH, POINTZ ; compare contents of location 3E16
CJNE A, #10, POINTW ; compare ACC with number 1010
CJNE R5, #34, LOOP
CJNE @R1, #5, GOINGON ; compare the number 5 with the contents of
                        internal RAM where R1 is pointing
```

The carry flag is altered by this instruction. *CJNE* can be the first part of a *greater-than/equal-to/less-than* test. If the second number of the com-



parison is bigger, there will be no carry. If the second number is equal to or smaller than the first, there will be a carry out. So it is possible to get a three-way branch as follows:

---

```

    CJNE A,X, NEXT__TEST
                JMP EQUAL
NEXT_TEST: JNC X_SMALLER
                JMP X_BIGGER
EQUAL:        .....
                .....
                JMP GO_ON
X_SMALLER: .....
                .....
                JMP GO_ON
X_BIGGER:    .....
                .....
                JMP GO_ON
GO_ON:       .....

```

---

With this combination, you can jump to one of three different places depending on the relative values in the accumulator and the variable X.

## DJNZ

This is a very handy instruction for iterative loops where the number of times to go around is set outside the loop and then counted down to zero. For example, you could have a loop to step a motor ten times, using a subroutine call (described next):

---

```

        MOV 03FH, #10    ; (set up for loop)
LOOP1:  DJNZ 3FH, GO_ON   ; decrementing contents of location 3F16
        (here holding the number 1010, so this program piece would loop for
        ten times)
        CALL STEP
        JMP LOOP1
GOZ_ON: .....

```

---

## Calls

A *CALL* lets the program flow go to a subroutine and come back when the subroutine is finished.<sup>11</sup> If you might *jump* to a piece of code from sev-

---

<sup>11</sup>If you are new to programming, you may want to jump ahead to Chapter 7 on functions (in Section III) to appreciate the reasons for subroutines.

eral places, you have no way to know which *jump* brought the flow to the code piece and no way to get back to where you left off. *CALLs* can come from multiple places. The program counter is pushed on the stack at the *CALL*, and a *RET* restores the program counter from the stack. Program flow resumes with the instruction just after the particular *CALL* that led to the subroutine.

## ACALL

*ACALL* unconditionally calls a subroutine located at the indicated address. The program counter, already set to the address of the next instruction, is pushed onto the stack (low-order byte first), and the stack pointer is adjusted. For this instruction the subroutine address (where the program flow will go) is obtained by combining the five high-order bits of the incremented PC, opcode bits 7 to 5, and the second byte of the instruction. The subroutine must start in the *same* 2K block of code space as the instruction following *ACALL*. No flags are affected.

---

*ACALL STEPPERROUTINE* assuming the next code line begins at address  $201D_{16}$ , *STEPPERROUTINE* begins at  $2500_{16}$ , and that SP is initially at  $95_{16}$ , this instruction leaves the stack pointer with  $97_{16}$ , leaves  $1D_{16}$  at internal RAM location  $96_{16}$ , leaves  $20_{16}$  at  $97_{16}$ , and leaves the program counter with  $2500_{16}$ .

---

Unless you are hand assembling, it is best to let the assembler figure this out and just use a symbolic address such as:

---

```

                                ACALL STEPPERROUTINE
                                .....
STEPPERROUTINE: .....
                                .....
                                RET

```

---

## LCALL

The long call can go anywhere in the 16-bit address range. The stack and program counter functions are the same as *ACALL* except that the full 16-bit address is found in the second and third bytes of the instruction.

---

*LCALL DISPLAY* assuming the next code line begins at address  $23F1_{16}$ , *DISPLAY* begins at  $24AF_{16}$ , and that SP is initially at  $38_{16}$ , this instruction

---

---

leaves the stack pointer with  $3A_{16}$ , leaves  $F1_{16}$  at internal RAM location  $39_{16}$ , leaves  $23_{16}$  at  $3A_{16}$ , and leaves the program counter with  $24AF_{16}$ .

---

## RET

The return puts the top two values from the stack into the program counter. It allows the flow to resume following the completion of a subroutine.

---

**RET** ; assuming the situation after the *ACALL* above, this would bring the stack back down to  $95_{16}$  and restore the program counter to  $201D_{16}$ .

---

## RETI

The *RETI* behaves like *RET* with the additional feature of automatically resetting some interrupt hardware to allow further interrupts of the same priority level.<sup>12</sup>

## No Operation—NOP

In a sense, this instruction does nothing, but its usefulness is in the time it takes to execute; you can use it in software time delays.<sup>13</sup>

## LOGIC INSTRUCTIONS

### ANL

ANDing Logical gives a 1 in a bit position only where both bits are 1. Notice that this instruction leaves the result in the accumulator!<sup>14</sup>

---

**ANL A, R6** ; *ACC* (for example,  $1101\ 1000_2$ ) with *R6* (for example,  $1000\ 1111_2$ ) gives result in *ACC* (here  $1000\ 1000_2$ )  
**ANL A, 25H** ; AND *ACC* with contents of location  $25_{16}$   
**ANL A, @R1** ; AND *ACC* with contents of on-chip RAM location pointed to by *R1*

---

<sup>12</sup>See Chapter 11 on interrupts for more details.

<sup>13</sup>The *NOP* can also be useful to hold some space in a string of machine instructions to be later replaced by other instructions. It is particularly useful when using breakpoints with a monitor program (see Chapter 9).

<sup>14</sup>I would have loved to describe the TTL logic functions that you can implement in hardware, but space restrictions prevailed. The parallels between discrete logic and the functions inside the CPU are exciting—you actually start to see that the CPU is not “magic.”

---

```
ANL A, #03H ; AND ACC with the number 3
ANL 25H, A ; same as ANL A,25H
```

---

## ORL

ORing Logical puts a 1 in a bit place if either bit is 1.

---

```
ORL A, R6 ; ACC (for example, 1101 10002) with R6 (for example, 1000
          11112) gives result in ACC (1101 11112)
ORL A, 25H
ORL A, @R1
ORL A, #03H
ORL 25H, A
```

---

## XRL

eXclusive oRing Logical gives a 1 in a bit position if one *and only one* bit is 1—if both bits are 1, it puts a 0.

---

```
XRL A, R6 ; ACC (for example, 1101 10002) with R6 (for example, 1000
          11112) gives result in ACC (here 0101 01112)
XRL A, 25H
XRL A, @R1
XRL A, #03H
XRL 25H, A
```

---

## CPL

The CoMplement Logical changes 0s to 1s and 1s to 0s.

---

```
CPL A ; start with ACC (for example, 1101 10002) leaves result in ACC
      (here 0010 01112)
```

---

## CLR

This CLears sets all bits to zero (the accumulator only).

---

```
CLR A ; sends 0s to all 8 bits of the accumulator
```

---

## Rotating

In addition to adding and inverting, there are ways to move data to the left or right, called *shifting* or *rotating*. A value of 0001<sub>2</sub> (1<sub>10</sub>) shifted left is 0010<sub>2</sub> (2<sub>10</sub>). Likewise 1010<sub>2</sub> (10<sub>10</sub>) shifted right is 0101<sub>2</sub> (5<sub>10</sub>). In binary math, a shift is a multiply or divide by two. The shifting can be done di-

rectly by logic hardware or it can be by wiring outputs to inputs one-over from the original position.

### RR, RL, RRC, RLC

With these instructions, the accumulator shifts by one place left (toward msb) or right (toward lsb). If the carry is included then the end bit goes into the carry and the carry goes around into the other end.<sup>15</sup>

---

RL A ; an 8-bit shift around toward the most significant bit; 1011 1100<sub>2</sub> becomes 0111 1001<sub>2</sub>  
 RR A ; 8-bit shift right, 1011 1100<sub>2</sub> becomes 0101 1110<sub>2</sub>  
 RRC A ; 9-bit shift right toward lsb; carry goes into msb; 1 1011 1100<sub>2</sub> becomes 0 1101 1110<sub>2</sub>  
 RLC A ; 9-bit shift left; carry goes into lsb; 1 1011 1100<sub>2</sub> becomes 1 0111 1001<sub>2</sub>

---

## BOOLEAN (BIT) INSTRUCTIONS

### Flags and their uses

Flags are single-bit variables you can directly manipulate<sup>16</sup> or that change indirectly because of executing instructions. In the 8051, there are a possible 128 bits that you can use as simple 1-bit variables<sup>17</sup> as well as bits within bytes of the SFRs.<sup>18</sup>

#### Instructions that directly affect flag settings

Instruction	Flag Affected		
	Carry (C)	Overflow (OV)	Auxiliary Carry (AC)
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, bit	X		
ANL C, /bit	X		
ORL C, bit	X		
ORL C, /bit	X		
MOV C, bit	X		

<sup>15</sup>The rotate left brings the top bit around into the bottom (or the carry position). C has *shift* instructions (>> and <<) that move bits left or right but do *not* bring them around. Instead the bits are discarded, and 0s fill in the space.

<sup>16</sup>In addition to the directly flag-related instructions shown in the table, you can directly affect the flags by byte-addressing the PSW register where they reside.

<sup>17</sup>Bit addresses 00 through 7F<sub>16</sub>, overlapping with internal byte addresses 20<sub>16</sub> through 3F<sub>16</sub>.

<sup>18</sup>Addresses 80<sub>16</sub> to FF<sub>16</sub>.

Directly addressable flags are useful for marking the occurrence of an event. Take a traffic light program; if someone has pushed a walk button, you can *set* (give a value of 1 to) a flag, and then *clear* (give a value of 0 to) the flag later at the right place in the cycle when the program turns the walk light on. The next time around the light cycle, if no one has pushed the button since the last walk cycle, the light need not be turned on again.

The individually affected flags are particularly useful for multibyte math operations. You can add the two least significant bytes *without* the carry (*ADD*) and then add the next 2 bytes *with* carry (*ADDC*) that was set by the first addition, on up to as many bytes wide as you want.<sup>19</sup> An example of 2-byte addition would be:

---

```

;two numbers in R6-R7 and 31H-32H; result in R6-R7
MOV A, R6
ADD A, 31H    ; with no carry
MOV R6, A
MOV A, R7
ADDC A, 32H   ; with carry set by first addition
MOV R7, A

```

---

#### Instructions that Indirectly Affect Flag Settings

Instruction	Flag Affected		
	Carry (C)	Overflow (OV)	Auxiliary Carry (AC)
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RRC	X		
RLC	X		
CJNE	X		

---

<sup>19</sup>Overflow helps with signed math operations. With addition it indicates a carry out of bit 6, but not bit 7, or reverse. For *signed* math, *OV* indicates that two positive numbers gave a negative answer or two negative numbers gave a positive result. With multiplication giving a 16-bit result, *OV* indicates that the answer has exceeded the lower byte and *B* must be taken into account. On the rare occasion when you might do decimal math (BCD notation), the auxiliary carry is used by the *DAA* instruction to decide whether to add 6 to the result.

Of the *indirectly* affected flags, the most significant are the three bits (*C*, *OV*, and *AC*) within the *PSW* that are affected by other instructions.

### CLR (bit)

This clears (sets to 0) the bit involved. It works on the 128 bits in the  $20_{16}$  to  $2f_{16}$  byte-addressed area as well as the bit-addressable SFRs.

---

```
CLR C      ; set the carry bit to 0
CLR ACC.7   ; set the msb of accumulator to 0
CLR P1.5    ; set the third-from-the-top bit of port 1 to 0
```

---

### SETB

The set makes the indicated bit a 1.

---

```
SETB C      ; set the carry bit to 1
SETB 20.3    ; set directly-addressable memory bit fourth up in first byte
               (byte address 20H—same as bit address 03H) to 1
SETB ACC.7    ; set msb of accumulator to 1
```

---

### CPL (bit)

This complements (changes 0 to 1 or 1 to 0) the bit.

---

```
CPL C      ; invert the carry bit—1  $\Rightarrow$  0 or 0  $\Rightarrow$  1
CPL P3.5    ; invert third-from-msb of port 3
```

---

### ANL (bit)

---

```
ANL C, ACC.5 ; AND carry and bit-5 of accumulator—result in carry bit
ANL C, /ACC.5 ; AND carry and complement of bit-5 of accumulator—
               result in carry bit; accumulator not changed
```

---

### ORL (bit)

---

```
ORL C, ACC.5 ; OR carry with bit-5 of accumulator—result in carry bit
ORL C, /ACC.5 ; OR carry with complement of bit-5 of accumulator—
               result in carry bit; accumulator not changed
```

---

**MOV (bit)**

These could be grouped with the other *MOV* instructions, but because they are bit-oriented, this seemed a better place to describe them.

---

```
MOV C, ACC.2 ; contents of third bit of accumulator (a 1 or a 0) is moved
              into the carry bit
MOV 20.3, C ; contents of carry is moved into bit-addressable RAM
              location 3 (20.3 is fourth bit of byte-address 20 which is the first byte
              of bit-addressable RAM)
```

---

**MATH INSTRUCTIONS**

The basic 8051 family has very limited math capability. Single-byte instructions can be put together in multi-instruction program pieces to do far more elaborate math.<sup>20</sup> Other than increment and decrement, all CPU arithmetic operations overwrite the accumulator contents with the new result.

**Addition**

**ADD.** This instruction does not bring in the carry bit with the least significant bit (lsb), but it does produce a carry-out result. It is instruction for the first step of a multibyte operation, but you may prefer to zero the carry bit and just use *ADDC* throughout any multibyte sequence.

---

```
ADD A, R5 ; add contents of R5 to ACC; result in ACC; overflow into carry
ADD A, 22 ; add contents of internal RAM location 2210 (1616) to ACC;
           result in ACC; overflow in carry
ADD A, @R0 ; add contents of internal RAM pointed to by contents of R0 to
            ACC; result in ACC; overflow in carry
ADD A, #22 ; directly add the number 2210 (1616) to ACC; result in ACC;
           overflow in carry
```

---

**ADDC.** This includes the carry bit in the addition.

---

```
ADDC A, R5 ; add contents of R5 with carry to ACC; result in ACC;
           overflow into carry
ADDC A, 22 ; add contents of internal RAM location 2210 (1616) with carry
           to ACC; result in ACC; overflow in carry
```

---

<sup>20</sup>The key to getting more elaborate math functions is the algorithm that can take simple math instructions of the microcontroller and do the math of statistics, trigonometry, and even calculus! At least one algorithm (several-byte addition) is shown in Chapter 8, but most of the coverage is in the companion book. I don't intend to solve differential equations on an 8051!



ADDC A, @R0 ; add contents of internal RAM pointed to by contents of R0, with carry, to ACC; result in ACC; overflow in carry  
 ADDC A, #22 ; directly add the number 22<sub>10</sub> (16<sub>16</sub>), with carry, to ACC; result in ACC; overflow in carry

---

## Subtraction

**SUBB.** The *borrow* flag is the inverse of the carry out from a normal adder. There is no instruction to subtract without the borrow, so you will have to clear the borrow, before beginning subtraction.<sup>21</sup> The borrow indicates that something too big was subtracted and you got a negative answer rather than what appears to be a very large positive number. For multibyte subtraction, work your way from the least-significant byte up to the most-significant byte—borrows along the way are acceptable. The answer is positive (and correct) as long as there is no borrow when the most significant bytes have been subtracted. Remember that this is *unsigned* math. Make any adjustments for negative numbers in software that you write. Signed 16-bit integer math is straightforward too, but I suggest you leave that for a C compiler to manage.

---

SUBB A, R5 ; subtract contents of R5 with borrow from ACC; result in ACC; underflow into borrow (carry); ACC = C9<sub>16</sub>, R5 holds 54<sub>16</sub>, carry is set then result is ACC = 74<sub>16</sub> and carry/borrow is cleared (positive result) but OV (bit 6 to 7 carry/borrow—used for signed math) is set  
 SUBB A, 22 ; subtract contents of internal RAM location 22<sub>10</sub> (16<sub>16</sub>) with borrow from ACC; result in ACC; underflow in borrow  
 SUBB A, @R0 ; subtract contents of internal RAM pointed to by contents of R0, with borrow, from ACC; result in ACC; underflow in borrow  
 SUBB A, #22 ; directly subtract the number 22<sub>10</sub> (16<sub>16</sub>), with borrow, from ACC; result in ACC; underflow in borrow

---

## Other Math

**INC and DEC.** These are symmetrical instructions except for the data pointer. Decrementing 0 or incrementing FF<sub>16</sub> will roll over to FF<sub>16</sub> or 0 respectively.

---

<sup>21</sup>Although you will not see the term *borrow* in the 8051 flag descriptions, it is the *carry* flag. Doubly confusing is the fact that this borrow flag function is just the inverse of the carry bit from a full adder doing subtraction by adding the complement. If a subtraction would produce a carry from the hardware, then the 8051's *carry flag* is *not* set!

INC A	DEC A
INC R2	DEC R5
INC 45H	DEC 3EH
INC @R0	DEC @R1
INC DPTR	no decrement equivalent exists

**MUL.** There is only one hardware-multiply, which produces a 16-bit result in the accumulator (low byte) and the B register (high byte). If the product exceeds 8-bits, the overflow flag is set. The carry flag is always cleared.

---

**MUL AB** ; ACC multiplied by contents of B register; result in ACC (low-byte) and B (high byte); for example  $B = 160_{10} (A0_{16})$ ,  $ACC = 80_{10} (50_{16})$ , result is  $12,800_{10} (3200_{16})$ ;  $B = 32_{16}$ ,  $ACC = 00_{16}$ , carry is always cleared,  $OV = 1$  indicating that result is greater than 8 bits ( $B$  holds part of the result)

---

**DIV.**  $B$  divides  $ACC$ , with the result in  $ACC$  and the remainder (not the fraction) in  $B$ . The carry and overflow flags are always cleared. Neither operation neatly supports multibyte extensions or signed math—certainly not floating-point math!

---

**DIV AB** ;  $ACC$  is divided by contents of  $B$  register;  $ACC$  holds integer part of quotient while  $B$  has the integer remainder; for example  $251_{10} (FB_{16})$  in  $ACC$  divided by  $18_{10} (12_{16})$  gives  $13_{10} (0D_{16})$  in  $ACC$ , a remainder of  $17_{10} (11_{16})$  in  $B$  and carry and  $OV$  are always cleared.

---

## Decimal Instructions

These instructions are seldom used except with BCD data. You might use them shuffling nibbles from a 12-bit A-D converter.

**XCHD.** This instruction exchanges the low-order nibble (4-bits) of the accumulator with the indirect addressed value. The upper nibble remains with the original location. It has to be an artifact left over from BCD math.

---

**XCHD A, @R0** ; example  $ACC$  holds  $3A_{16}$ ,  $R0$  points to location holding  $F0_{16}$ ; result  $ACC$  holds  $30_{16}$ ,  $R0$  points to location holding  $FA_{16}$

---

**SWAP.** This instruction reverses the places of the upper and lower nibble of the accumulator.

---

SWAP A ; example ACC starts with  $34_{16}$ ; ends up with  $43_{16}$

---

**DAA.** This Decimal Adjusts the Accumulator. If you are adding packed BCD digits (a byte holding two 4-bit BCD numbers) and the AC flag is set or the current values of either nibble exceed 9, then this instruction adds 00H, 06H, 60H, or 66H as needed to bring the digits back into BCD form.

---

ADDC A, R3 ; assume ACC holds  $56_{16}$  (packed BCD for  $56_{10}$ ) and R3 holds  $67_{16}$  (packed BCD for  $67_{10}$ ); result is  $BE_{16}$  with carry and auxiliary carry set

DAA ; given first instruction the result is  $24_{16}$  in ACC with a carry set (indicating packed BCD for  $24_{10}$  with an overflow or  $124_{10}$ —the decimal result of  $56_{10} + 67_{10}$ )

---

This will not do a magic hex-to-decimal conversion and *is of no use for decimal subtraction*. It is preferable to do math in binary (hex) form and convert to or from decimal only when human input or output is involved.

## WRAPPING UP THE INSTRUCTION SET

### Writing Assembly Language

We have now explored all the 8051 assembly instructions. I expect by now you are overwhelmed—the key is, *you don't have to use all the instructions all at once, and some you may never use*. If you are going to write in assembly, start with a few simple programs to move data around and do some simple math. You will need to become familiar with the tools from Chapter 9 to do anything, but a good exercise is to load some memory locations and then see what happens as you run the instruction that affects them.

If you are overwhelmed by the requirement of putting these instructions together to make things happen, relax and go on to examples in the next few chapters. While the focus there is on the C language constructs, it also includes the assembly language equivalents. Anything you can do in C, you can also do in assembly.

## EXERCISES IN ASSEMBLY LANGUAGE

1. Write a program that will load each register and memory location with the contents shown below.

Starting Conditions					
Registers		Internal Memory		External Memory	
A	23H	35H	74H	20DDH	12H
B	04H	36H	82H	2100H	13H
SP	40H	0AH	19H		
DPTR	20DDH				
PSW	81H				
R0	35H				
R1	36H				
R2	15H				
R3	5CH				

Simulate (using DS51 described in Chapter 9) or load and run your program (using techniques also described in Chapter 9 to verify that your program sets up all initial conditions properly.

2. For each case shown below, *assuming the starting conditions above*, work out all the registers that will be modified and their new contents. Then list the *hand-assembled* hex bytes necessary to carry out the instruction. The first instruction is completed as an example.

Next check your predictions by loading your program, byte by byte.

Re-establish the initial conditions after each step using the program of part 1.<sup>22</sup>

---

<sup>22</sup>The first problem might be run as follows (in parentheses are the commands for the EET monitor program):

Make sure all the registers and memory locations shown are set to their starting conditions by using the monitor commands to set each location (or by executing the setup program above).

Load the following op code and data bytes at the end of your setup program

Location	Contents
2021H	ESH
2022H	0AH

Enter the Run command. (Type G and set the starting address to 2000<sub>16</sub>. Do not press Return.)

Enter the break address. (Press the comma key and enter a breakpoint address of the *Next* location after the op code and data bytes in the instruction you are testing. In this example it is 2023<sub>16</sub>. Now press return.)

After the breakpoint is executed and the registers are displayed, consult the register display or use the display command to examine other memory spaces as necessary to assure that your predictions were accurate.

Reload the *next* instruction code that you need to test (in place of the last one) at the end of your setup program and again run the program from 2000<sub>16</sub>.

Instruction		Predicted	Actual
MOV A, 0AH	Registers Modified, Contents: Assembled Code:	A = 19H  E5 0A	
MOV SP,#7	Registers Modified, Contents: Assembled Code:		
MOV @R0, A	Registers Modified, Contents: Assembled Code:		
MOV R3, A	Registers Modified, Contents: Assembled Code:		
MOV A, #0AH	Registers Modified, Contents: Assembled Code:		
MOV R3, 0AH	Registers Modified, Contents: Assembled Code:		
MOV 36h, 35h	Registers Modified, Contents: Assembled Code:		
MOV @R1, #0AH	Registers Modified, Contents: Assembled Code:		
MOV A, @R0	Registers Modified, Contents: Assembled Code:		
MOV @R1, 0AH	Registers Modified, Contents: Assembled Code:		
MOV DPTR, #3000H	Registers Modified, Contents: Assembled Code:		
MOVC A, @A+DPTR :	Registers Modified, Contents: Assembled Code:		

(continued)

Instruction		Predicted	Actual
MOVX @DPTR, A	Registers Modified, Contents: Assembled Code:		
MOV ACC.4, C	Registers Modified, Contents: Assembled Code:		

3. Predict the results of each instruction below assuming each instruction starts with the initial conditions given. Hand assemble each instruction into machine code. All starting conditions are the same as before. Then enter and run the instructions as in the earlier exercise.

Instruction		Predicted	Actual
ADD A, #0AH	Registers Modified, Contents: Assembled Code:		
ADDC A, 03H	Registers Modified, Contents: Assembled Code:		
SUBB A, 08H	Registers Modified, Contents: Assembled Code:		
SUBB A, #92H	Registers Modified, Contents: Assembled Code:		
SUBB A, @R0	Registers Modified, Contents: Assembled Code:		
MUL AB	Registers Modified, Contents: Assembled Code:		
DIV AB	Registers Modified, Contents: Assembled Code:		
SWAP A	Registers Modified, Contents: Assembled Code:		

(continued)

Instruction		Predicted	Actual
XCH A, B	Registers Modified, Contents: Assembled Code:		
XCHD A, @R1	Registers Modified, Contents: Assembled Code:		
CPL C	Registers Modified, Contents: Assembled Code:		
ANL 03H, #0F0H	Registers Modified, Contents: Assembled Code:		
ORL B, #20H	Registers Modified, Contents: Assembled Code:		
XRL PSW, #80H	Registers Modified, Contents: Assembled Code:		
PUSH PSW	Registers Modified, Contents: Assembled Code:		

4. Load each of the following instructions like you did in the previous exercise. Also place a *JMP 0030H* (02, 00, 30) instruction at address 2080<sub>16</sub>–2082<sub>16</sub>. If the jump *does* occur, program flow will reach the return to monitor (*JMP 30H*). If the jump does *not* occur, it will reach the breakpoint you set after the instruction you are testing.

Instruction	Machine code	Did the jump occur?	What address stores the next op code?
JZ 2080h			
JNC 2080H			
CJNE @R1, #28H, 2080H			
DJNZ R2, 2080h			

Instruction	Machine coda	Did the jump occur?	What addraaa stores the next op code?
JB PSW.2, 2080H			
Instruction	Machine coda	What other registers have been affected? new contents?	What address stores the next op code?
LCALL 2080H			

5. Disassemble the program contained in this hex file (Intel format) and figure out what the program does.

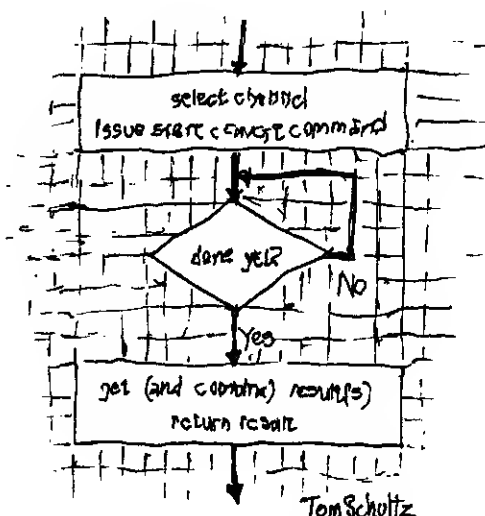
:0E20000078107455903010F0A3D8FC02003018
:00000001FF



# 4

## Two Languages

This book features the two languages, C and assembly, that developers most often use to write instructions for microcontrollers in the 8051 family. This chapter covers how numbers are described and stored as well as what to do with them. The emphasis is on the operations that are the specialty of microcontrollers—integer (no fractions) math, logical operations, and bit operations.



### WHY THESE TWO?

The 8051 microcontroller family, to my knowledge, is supported in only five languages—assembly, PL/M, C, Forth, and BASIC. Some programmers use BASIC quite a bit—usually with an *interpreter*.<sup>1</sup> One BASIC com-

<sup>1</sup>*Interpreted BASIC* is routinely available on PCs and may have been your first programming language. It is for just that purpose—a *basic* introduction to programming. There are few “messy” details when you start. The use of a new variable name automatically defines it as a variable for the rest of the program. Errors stand out as soon as you complete each line, rather than showing up when you finish writing the program. Also, it is easier to make changes line by line. There are, however, two definite reasons why interpreted BASIC *may not* be a good choice for embedded applications.

First, by its *interpreted* nature, ordinary Interpreted BASIC is slow—during the running of the program the interpreter converts each line to machine code as it comes up. The process of getting from English to machine code is a complex one and takes lots of computing time.

Second, to simplify the use of variables, all variables are full floating-point values. Adding  $2 + 2$  is a full floating-point math operation. This requires a complex set of machine instructions, so running time gets long. Even with compiled BASIC, if the program only sup-

pilers can produce compiled code.<sup>2</sup> I do not discuss Forth and PL/M at all.<sup>3</sup> C and assembly are the only languages used in this book. Once you have written a program, Chapter 9 goes into the process of getting your program (source code) transformed to an actual program running on an 8051.

C is the language originally used to write the UNIX operating system and for a long time it remained closely tied to UNIX. It is a *structured language*, and can produce compact source code. Braces { } rather than words mark out the structure and the language relies heavily on symbols seldom used in normal writing. You can control many machine-level functions without resorting to assembly language. You can *so* condense C, however, that the next programmer may have to study your program for some time to figure out what is going on! Only with careful attention can you write programs that nonprogrammers can understand. C is much easier to write than assembly language, because the development software manages the details. It has been available for 8051 microcontrollers since about 1985 and there are now several companies selling 8051 C compilers. Not all these compilers produce efficient code taking best advantage of the strange architecture of the 8051. Where the C examples that follow become compiler specific, it is Keil's version 5 compiler that is in view. A free version of this compiler is available from Keil as well as their more extensive professional versions.

---

ports floating-point math, resulting programs will be slow and large. One or two versions of the language *interpreter* exist for the 8051 family. You can obtain them, I believe, as public-domain code or you can buy them pre-programmed in an 8052. Although interpreted BASIC is quite suitable for applications where the ease of programming is more important than efficiency or speed, I will not discuss it further in this book.

<sup>2</sup>Compiling is discussed in Chapter 9. One company, Systronics (see Appendix A6) markets a *compiled BASIC* for the 8051 family. The interpreting then happens ahead of time much like a compiler processes the C code, and you can download or burn the resulting code in EPROM. I believe the current version uses integer variables, so it should be much faster running than conventional interpreted BASIC.

<sup>3</sup>PL/M has fallen into disuse. It is a proprietary Intel language that has been available for their processors beginning with the 8080. It looks much like PASCAL, but has roots in PL-1. Like C, it is a *structured* language, but it uses a more readable arrangement of key words to define the structure. PL/M is a high-level *assembly* language in both a negative and positive sense. You can have very detailed control of the code generated, but, for the 8051 family, PL/M has no support for complex mathematics, floating-point variables, or trigonometric functions. The language has worked well, but the cost of the compiler from Intel combined with the intense development that has continued to go into C has rendered it obsolete. See Appendix A2 for PL/M to C conversion hints.

*Assembly language* for the 8051, described in detail in Chapter 3, is much like other assembly languages.<sup>4</sup> Although it is an annoyance to learn *another* assembly language, the process is not difficult if you are already proficient in one. The instruction set is a bit more “powerful” than that of the first-generation microprocessors, but it is the different memory regions inherent in the 8051 family that complicate things. The move, math, logical, and branching instructions are generally similar to those of most other processors.

## VARIABLES

**Data Types for Variables**

Data Type	Size	Range
bit	1 bit	0 or 1
unsigned char	1 byte	0 to 255
unsigned int	2 byte	0 to 65535
(signed) char	1 byte	-128 to +127
(signed) (int)	2 byte	-32768 to +32767
(signed) long	4 byte	-2147483647 to +2147483646
unsigned long	4 byte	0 to 4294967295
float	4 byte	6 decimal digits e-37 to e +37
double	8 byte	10 decimal digits

Because computing starts with numbers, you need to understand how they are stored and represented.<sup>5</sup> The choice of variable type or data type is far more critical with the 8051 than with math-oriented computers. Machine instructions (and assembly language) directly support only the first two types in the table. Although a high-level language operation may look quite simple in your program, *a whole series* of machine instructions is necessary to handle the manipulation of the more complex variables. Using floating-point variables in particular will add considerably to the computing time and

<sup>4</sup>See Appendix A2 for language-switching hints from 8080 assembly.

<sup>5</sup>Later you will see how letters and words are represented by numbers to allow computers to do very nonmathematical things. In one sense the term *computer* is a misnomer for most activities done by today's machines.

program size.<sup>6</sup> When such precision is *necessary*, it is easy to let C automatically include the library functions. In assembly language, the overhead of any unnecessary use of a floating-point variable is obvious because *you* must write the functions to handle the variables. If you accidentally make your C compiler pull in large blocks of code from libraries, you will later discover your program is slow running and perhaps too large to fit in the available code space.

One variable type is the *bit*. A bit is either 1 (“true”) or 0 (“false”). Bit variables that use the 8051 instructions are put in on-chip RAM. Although 8051 instructions support the bit hardware directly, in C the use of those bits is an extension to the standard (portable) language.<sup>7</sup> The C language is not inherently very bit oriented—there is not even a binary notation for constants. You have to make do with hexadecimal notation. Most of the 8051 compilers add some way to define and use the bit-addressable features of the 8051 family, but technically those *extensions* make the language not fully portable.<sup>8</sup> Ultimately, you must decide how machine-independent your code needs to be. When the 8051 C compilers are most machine-independent, they produce the largest, most inefficient code!

**SIGNED AND UNSIGNED:** If you use both types you may pull in two forms of the library functions, which will use up more code space. If speed is important and negative numbers aren't needed, make everything *unsigned*.

*char* variables in C are 8-bit values, which fit ideally in the 8051 because it handles 8-bits at a time. They have values from 0 to 255 (*unsigned*) unless they are *signed*. Then the most significant bit, *msb*, is the sign. A 1 represents a negative, so signed and unsigned representations are the same

---

<sup>6</sup>I think the misguided desire to use floating-point variables comes from using scientific pocket calculators for early courses. When the speed of operations can be hundreds of milliseconds and still satisfy the user, the gyrations involved in variable representation are of little concern. For more on this question of philosophy, see the section on “real-time” in the multitasking section.

<sup>7</sup>The bit *fields* within an integer for ANSI-standard C are quite a different thing and are very implementation-dependent.

<sup>8</sup>You can't directly address bits on an x86, for example.

for 0 to  $127_{10}$ . The computer represents negative numbers by 2's complement notation, which makes  $-1_{10}$  to be  $1111111_2$  and  $-2_{10}$  to be  $1111110_2$ . Interestingly enough, that is just what would happen if a binary counter was at 0 and you subtracted 1 or 2 from it.

**UNSIGNED CHAR:** Use *unsigned char* variables whenever possible since the hardware handles them directly. Use bit variables for the same reason. *Signed char* variables take only 1 byte but any manipulation of them will pull in extra code to test the sign.

*int* (integer) variables in C are 16-bit values. Unlike the 8080 and x86 families, they are stored with the *most significant* byte at the *lower* address. *Signed* values again have the msb as the sign bit and use 2's complement notation. Similar to the *int* variables are the 4-byte (32-bit) *long* variables.

More complicated exponential representations in C are the *float* and *double*. They have both sign and magnitude for the exponent and the mantissa. ANSI C does not specify the implementation of floating-point storage, so the arrangement of the sign and exponent parts could vary among compilers. With float and double, for any math operations, the C compiler brings in library functions that have varying degrees of efficiency (depending on the compiler). Indeed, this is one field where marketing fights the "compiler war" most intensely. For most hardware-intensive control applications, you do not need floating-point manipulation!

## SHORTHAND: #DEFINE OR EQU

Many C programmers choose to develop shortcuts to save typing. You can easily do this with *#define* expressions at the top of the listing. After a few examples using the official C words, I will switch to my favorite abbreviations to shorten the listings. Specifically, in C, I will use *uchar* for *unsigned char* and *uint* for *unsigned*. I will usually show the definition lines in the examples, but realize that the body of the example will be nonstandard (and more compact!).

In assembly, you can do the same with an *EQU* line. You can substitute a single word for a longer expression.

## MEMORY SPACES

As was hinted in the last chapter, at least three different memory spaces can have the same address. First, there is *code* space, for program instruction code and other unchanging (constant) information. This would eventually go into an EPROM. There are no instructions to write *to* code space because the program should not modify itself. Code space is also the logical place to keep canned messages used for interaction with a user/operator.

The second memory space is *internal* RAM (the place for *data* variables). It is between 64 and 256 bytes long depending on the particular processor (the original was 128), and is always part of the microcontroller chip. That is not very much memory, but there is a rich set of machine instructions to access it. The *internal data* memory is a good place to “park” variables temporarily for computations, as well as a place to keep frequently used variables.

**ON-CHIP RAM:** Used on-chip RAM for frequently used variables. In C you can either use a *small* memory model that puts all variables there by default or, depending on the compiler, force them there by the use of *idata* or *data* keywords. In assembly you simply avoid the *MOVX* instruction and the *XSEG* designations.

Finally, there is external *xdata* memory that is not on the 8051 chip itself. It is commonly 2K to 64K bytes in one added chip.<sup>9</sup> The machine instructions to access this memory must bring the value into internal memory before putting it to use—a process involving several machine instructions in itself—and then return the result to external memory. External memory is the place for less-frequently used variables and for collected data waiting for processing or forwarding to another computer.

---

<sup>9</sup>The *external* memory in a very few members of the family is physically part of the chip, but you must still move those values into data space before you can use them.

### Memory Allocation

```

10 define PORTA XBYTE[0x4000];
11 bit flag1; /*assignment example*/
12 code char table1[] = 1,2,3,"HELP",0xff;
13 idata unsigned int temp1;

```

**CHARACTER STRINGS: SINGLE AND DOUBLE QUOTES:** In C it is easy to confuse them. Single quotes do not add the end of message character (usually a zero) to the character string and can only be up to four in length (a *long* variable).

**COMMENT LINES:** Probably the most elusive error for C programs is the failing to end a comment line properly. A comment ends when the compiler encounters a `*/`. If the `*/` gets left off, the following lines become comments until the (correct) end of the *next* comment. Usually the error comes near the start and misses some of the variable definitions, producing strange errors all through the program. The color feature of the editor that comes with this book makes it quite easy to detect the problem, because the comments are *green*—if you find some code looking green, you have messed up a comment! The other way to detect the problem is to look at the *list* file and find program lines where the indentation depth numbers are missing.

Two other seldom-used memory types that are available in assembly language are *idata* and *pdata*. The former is indirectly addressable internal

<sup>10</sup>Some machine-specific additions to C show here. The `#define` for *PORTA* is specific to Keil/Franklin—other compilers use different words than *XBYTE*. This defines a *fixed* location (here an 8255 from an example on page 16).

<sup>11</sup>This defines a single-bit variable (which can only be in internal RAM).

<sup>12</sup>This defines a group of 9 bytes (in ROM/EPROM) that will begin at an address later assigned to *TABLE1*. "help" includes an end-of-string character (a 0). The four bytes for *HELP* are stored in ASCII code where 0 through 9 are 30<sub>16</sub> through 39<sub>16</sub> and (upper case) alpha characters go from A at 41<sub>16</sub>.

<sup>13</sup>This sets up a 2-byte space for storage in internal RAM above address 127<sub>10</sub>.

memory (above 127<sub>10</sub>).<sup>14</sup> The latter is *xdata* with a one byte address where *P2* is being carefully preserved for I/O. Both are of little interest for most expanded-memory applications because it is difficult to even *buy* a 256-byte memory chip. It *could* apply to an application that only added off-chip I/O devices in that memory space.

The allocation of memory space is different for assembly, as shown below. The example shows *relocatable* segments. Beginning programmers may be encouraged to make *all* their code *absolute* (requiring no linking) using the *AT* directive.<sup>15</sup>

---

#### MEMORY ALLOCATION

```

MYBITS SEGMENT BIT
MYROM SEGMENT CODE
MYIRAM SEGMENT DATA
MYIRAM SEGMENT IDATA
RSEG MYBITS
16     FLAG1: DBIT 1
RSEG MYROM
17     TABLE1: DB 1,2,3,'HELP',0FFH
RSEG MYIROM
18     TEMP1: DS 2
XSEG AT 6000H
19     PORTA: DS 1
20     END

```

---

<sup>14</sup>This indirectly addressed memory space is accessed by first setting the values of *R0* or *R1*, with *@R0* or *@R1* as pointers. You can indirectly address the entire 256 bytes, but if you directly address values above 127 you access the special function registers (SFRs) rather than the high memory.

<sup>15</sup>Chapter 8 covers in more detail many aspects of managing memory space as well as details of using assemblers and compilers.

<sup>16</sup>While it is possible to use an absolute memory location in assembly by just addressing a specific location by your instruction, it is safer to let the linker/locator utility assign addresses to avoid overlap or wasted space. In this first case, just like the C example before it, because *MYBITS* was earlier defined as a *BIT* segment, this line defines a single bit (which will be located in internal RAM by the linker/locator program).

<sup>17</sup>This is a group of 8 bytes in ROM, because *MYROM* was first defined as a *CODE* segment.

<sup>18</sup>This is a block of 2 bytes to be located in indirectly addressable internal RAM.

<sup>19</sup>Again, this sets up the label for a fixed location on external memory space—probably an I/O port. Note that this is an *absolute* (not *relocatable*) segment, so the address is defined with the *AT* directive and no segment name is required.

<sup>20</sup>All assembly programs must finish with an *END* directive.



## PORTS

To have realistic embedded examples I will use both on- and off-chip parallel I/O ports.<sup>21</sup> It takes special treatment to persuade the C compiler to put a variable at a fixed location. C has *include* files that define the SFRs of the 8051 members and the assembler, by default, knows the definitions for the basic 8051 port names.

**PORT USAGE:** The internal ports of the 8051 family are bit-addressable while most external add-on ports are only byte-addressable. In assigning ports to hardware, where there is a choice, put individual control lines and single-bit I/O signals on *internal* ports and byte-wide I/O on *external* ports. That will greatly simplify the addressing and avoid a lot of the program logic involved in masking bits.

## EXAMPLE: SWITCHES TO LIGHTS

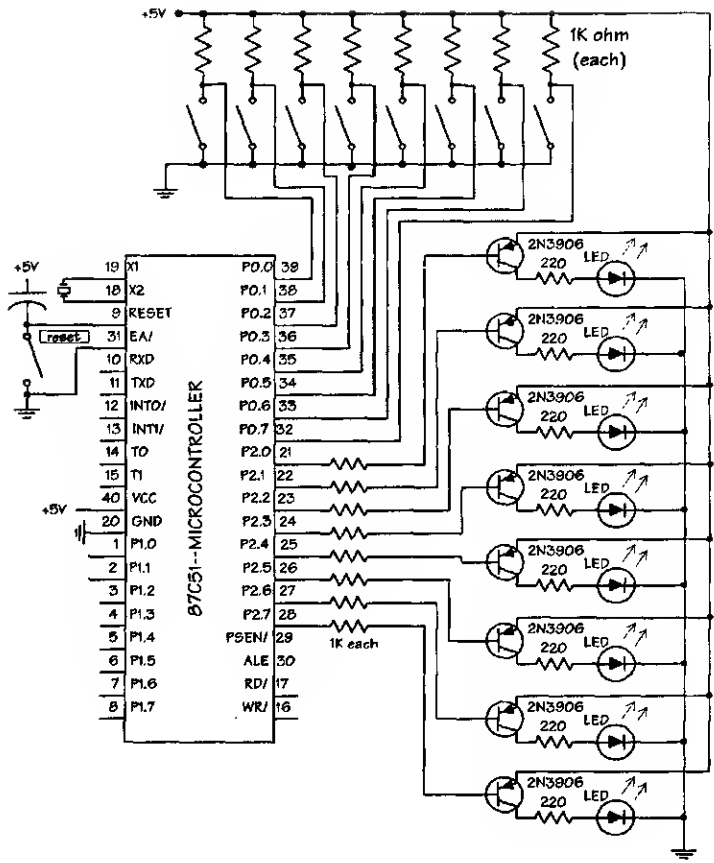
I start with a somewhat complicated example that could work on the hardware shown here.<sup>22</sup> This program runs on an 8751 (having on-chip EPROM). It reads in from 8 switches connected to (on-chip) *P0*, stores the readings in a 10-byte array, shows the most recent reading on 8 LEDs connected to *P2*, and waits 1/10 second to repeat the process.<sup>23</sup>

---

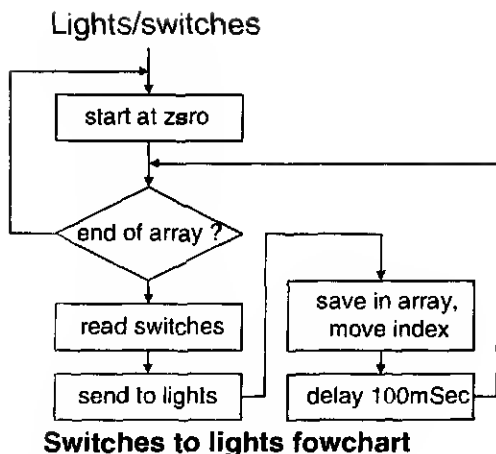
<sup>21</sup>The 8051 has no specific I/O instructions—the ports are just memory locations, which happen to show up as signals on external pins. There are four on-chip 8-bit I/O ports in the basic 8051 members (*P0*, *P1*, *P2*, and *P3*), but, often, expanding the EPROM or RAM space uses up 2½ of them. This was discussed earlier (pages 22–23) as well as the addition of I/O ports off-chip (pages 43–45).

<sup>22</sup>Chapter 9 goes into details of program compiling and linking and Chapter 3 went into some of the hardware design issues.

<sup>23</sup>Chapter 5 will discuss the millisecond time delay.



Switches to lights schematic




---

### Switches to Lights (Condensed Style)

```

#include <reg51.h>
void msec(unsigned int);
void main(){
    unsigned char array[10]; unsigned char i;
    while (1){
        for (i=0; i<=9; i++){
            array[i]=P2=P0;
            msec(100);}}
  
```

---

<sup>24</sup> With C there are a variety of commonly accepted ways to arrange the braces ( { } ) as well as a variety of indentations. The separation into lines and the indentation are not important to the compiler, but they can help the *readability* greatly. This expanded version is quite correct and is preferred by some who emphasize the structure of the language, but it becomes difficult with the deep indentation to fit lines on 80-column pages when the nesting becomes deep. The style of the condensed version is quite compact but raises complaints among some programmers, so I will use the last style throughout this book. For long blocks, the various braces by themselves can be confusing to type at the right nesting depth. If you carefully observe the shallow (two-space) indentation, the structure of the condensed form is easily visible. Obviously, if you totally ignore indentation depth, *any* system will be difficult to follow!

<sup>25</sup> The header file (or include file) defines all the registers and ports internal to the 8051. Actually this example only uses the definitions for *P0* and *P2*, but there are header files for a great number of different family members and there is nothing in them that couldn't be written out instead.

<sup>26</sup> The *msec* function is only *prototyped* here. The actual function would be found in some other module and this line only identifies how parameters would pass to and from the function. I discuss these details in the pages on scope of variables (Chapter 8, page 178). The delay function is discussed in Chapter 5.

---

```

Switches to Lights (Expended Style)24
25 #include <reg51.h>
26 void msec(unsigned int);
    void main()
    {
        unsigned char array[10];
        unsigned char i;
27     while (1)
        {
28         for (i=0; i<=9; i++)
            {
29                 array[i]=P2=P0;
                    msec(100);
            }
        }
    }

```

---



---

```

Switches to Lights (This Book's Style)
#include <reg51.h>
void msec(unsigned int);
void main(){
    unsigned char array[10];
    unsigned char i;
    while (1){
        for (i=0; i<=9; i++){
            array[i]=P2=P0;
            msec(100);
        }
    }
}

```

---



---

<sup>27</sup>An endless loop can be either a *while(1)* or a *for(;;)*, discussed in Chapter 5. For readability, either one is often defined as *forever*.

<sup>28</sup>*i* starts at 0 (points to the first element of the array), keeps on looping while *i* is less than or equal to 9, and steps *i* up by 1 each time. Because of the *while(1)*, it will begin overwriting the array after it has filled it the first time, so the array will always hold the most recent ten readings.

<sup>29</sup>The two equals evaluate from right to left—*P0* (the switches) is assigned to *P2* (the lights) before it is put into the array.

**DEEP INDENTATIONS:** It is generally unwise to use indentations over four spaces in C because the successively deeper indentations for nested blocks will force the comments to fold around on the list file if you print on an 80-character (8 1/2" wide) printer. Use a depth of two, I suggest, as is shown in most of the examples.

**TABS:** There can often be a problem with tab characters in a source file where the editor and the printer have different tab stop settings. Reconfiguring printers is often messy or out of your control. Rather than that, consider replacing tabs with spaces (which are universally understood). Some editors have a feature that will substitute spaces whenever you hit a tab as well as features that will follow a carriage return with enough spaces to bring the start of the next line under the start of the previous one. That is very handy for nested indentation (which is a must for any good structured program).

---

#### SWITCHES TO LIGHTS(TIGHT STYLE)<sup>30</sup>

<sup>31</sup>EXTRN CODE (MSEC)

<sup>32</sup>MYCODE SEGMENT CODE

---

<sup>30</sup>Two different styles for assembly programming are presented. The question of style for assembly is open to debate. Some would argue that this indentation is too shallow for the labels. Others would argue that the comments could be slid over to the end of the mnemonics. The assembler has no preference. It is a matter of personal taste as to how deep one indents and whether one keeps labels, mnemonics, and comments vertically aligned. In order to save space, I will use the tighter style in this book.

<sup>31</sup>The details of *MSEC* are discussed near the end of Chapter 5 and the details of combining program pieces from separate files is the subject of Chapter 8. For now it is included for completeness and can be thought of as "magic" to get a delay between readings.

<sup>32</sup>The segment lines define what type of memory space is to be used in linking the *RSEG* (relocatable segments). Here the array will be located in internal RAM and take up 10 bytes. The *MYCODE* segment is put in code space (EPROM). As written, after assembling, this code and data will need to be *located*, which is the preferred method when writing with multiple program modules. It is possible to include an *ORG* (originate or origin) instruction to direct the assembler to fill in specific jump addresses and put together an *absolute* piece of code. Such code would automatically "own" that space, and the link/locate utility program

```

MYDATA SEGMENT DATA
RSEG MYDATA
    ARRAY: DS 10
RSEG MYCODE
33  START:MOV R0,#ARRAY ;SET ARRAY PTR.
    AGAIN:MOV ACC,P0
    MOV P2,ACC
    MOV @R0,ACC ;STORE IN ARRAY
    MOV R2,#0 ;HIGH BYTE
    MOV R1,#100 ;LOW BYTE
34  LCALL MSEC
    INC R0 ;POINT TO NEXT LOC.
35  CJNE R0,#ARRAY+10,AGAIN ;END?
    SJMP START ;RESET TO START
END

```

---

#### ECHO SWITCHES TO LIGHTS(WIDE STYLE)

```

EXTRN CODE (MSEC)
MYCODE SEGMENT CODE
MYDATA SEGMENT DATA

RSEG MYDATA
    ARRAY: DS 10

RSEG MYCODE
    START:  MOV R0,#ARRAY           ;SET ARRAY PTR
    AGAIN:  MOV ACC,P0
            MOV P2,ACC
            MOV @R0,ACC             ;STORE IN ARRAY
            MOV R2,#0               ;HIGH BYTE

```

---

would have to use other space for code from *relocatable* modules. With multiple modules to combine in multitasking applications, it can be an annoyance keeping several absolute modules from overlapping. Chapter 8 discusses modular programming in more detail.

<sup>33</sup>The first *MOV* (move) instruction puts the starting address of *ARRAY* into the internal register *R0*. As *R0* increments, it will point to successive locations in the array.

<sup>34</sup>The time delay is not defined here and might involve a simple delay loop or the internal timers of the 8051 family. Software for timers appears in Chapter 11, page 285.

<sup>35</sup>When *R0* reaches *ARRAY+10*, the *CJNE* (compare and jump if not equal) instruction will bring program flow back to *START*. There *R0* will be reset to point to the first byte of array again.

```

MOV R1, #100                ; LOW BYTE
LCALL MSEC
INC R0                      ; POINT TO NEXT LOC.
CJNE R0, #ARRAY+10, AGAIN   ; END?
SJMP START                  ; RESET TO START
END

```

---

If you are curious about *how* C performs the operations, for this example, the assembly code it produced is shown next. You can request the *code* option for any C compilation, but you would usually include it only when very specific details are in question.<sup>36</sup>

---

**Assembly Code Resulting From C**

```

?C0001: CLR  A
        MOV  i, A
?C0003: MOV  A, i
        SETB C
        SUBB A, #09H
        JNC  ?C0001
        MOV  R7, P1
        MOV  P3, R7
        MOV  A, #array
        ADD  A, i
        MOV  R0, A
        MOV  @R0, AR7
        MOV  R7, #064H
        MOV  R6, #00H
        LCALL _msec
        INC  i
        SJMP ?C0003

```

---

## BITWISE LOGICAL OPERATORS

Control applications often use bitwise logical operations rather than arithmetic. With external input and output ports, it is often desirable to read or change one bit of a byte without affecting the other bits. Perhaps this one bit

---

<sup>36</sup>For example, in this case I used it to discover that the reversal of the order of assignment to *P2* and *array(i)* had no effect on code efficiency. For this book listing, I stripped off the actual machine code values with an editor but they would usually be to the left of the assembly listing as at the start of Chapter 2 (page 10). Keil/Franklin C puts the code listing separate from the high-level source whereas some other C compilers put the resulting assembly instructions interleaved with the C instructions.

turns a motor on or off while the other bits of the port activate warning lights or tell an A-D converter to start a conversion. Particularly with multitasking, it may be impossible to predict in advance what the *other* bits of the port will be. Some ports are bit addressable (those directly on the 8051 family chips, for example), but most add-on ports respond only as entire bytes. This is where *bitwise* logical operators come into play. The next table lists them for both languages.<sup>37</sup>

Bitwise Operators		
Logical operation	Assembly instruction	C
NOT	CPL A	~
AND	ANL A,#	&
OR	ORL A,#	
EXCLUSIVE OR	XRL A,#	^

Assume in the next examples that *PORTA* is an external byte-addressable port needing the third bit from the bottom (bit 2, because counting starts with bit 0) set high and needing bit 6 set low *without affecting any other bits*.

---

```

Bit Set and Clear
38     extern xdata unsigned char PORTA;
      void main(void) {
39,40     PORTA=(PORTA & 0xbf) | 0x04;
      }

```

---

<sup>37</sup>C makes a sharp distinction between *bitwise* logical operators and *logical* (true/false testing) operators. The latter only produce a true (ff<sub>16</sub>) or false (00<sub>16</sub>) result.

<sup>38</sup>This line declares the existence of a variable named *PORTA*, defined in a separate module, that would set the port at a specific address. In this example, it is intended to have an assembly language module handle such details as will be discussed in Chapter 8. If you would avoid this, the *XBYTE* approach allows direct addressing of ports without linking in another module.

<sup>39</sup>Hexadecimal constants start with 0x. There is no notation for binary numbers in C, so they must be expressed in hex or octal (without the x, a leading 0 (zero) on an integer constant means octal!). It is possible, in defining a constant, to force a definition of “long” or “floating” by appending a suffix of *l* or *f*.

<sup>40</sup>The & and | associate from left to right. The parenthesis are unnecessary, but it is always safer to include them.



**BIT SET AND CLEAR**


---

```

PORTA EQU 6000H
41  CSEG AT 2000H
    MOV DPTR,#PORTA
    MOVX A,@DPTR ;GET PRESENT READING
    ANL A,#10111111B ;SET BIT 6 LOW
    ORL A,#00000100B ;SET BIT 2 HIGH
    MOVX @DPTR,A ;OUTPUT NEW VALUES
    END

```

---

**CONFUSING BITWISE AND LOGICAL PRECEDENCE:** When a program waits for a bit to change (someone pushes a button), the *masking* may be done wrong.

*while (PORTA & 0x8 > 0)*

would seem to stop when the fourth bit goes low, since the *>* has precedence over the *&*, the right side evaluates as true (1 or FF), and there is really no testing for the intended bit.

**ROTATE AND SHIFT**

In addition to the operations already mentioned, two bit-related operators rearrange a byte. First is the *rotate*. If you think of a byte as a *collection* of bits from the left (*msb*—most significant bit) to right (*lsb*—least significant bit) order, you can see that a *rotate right* moves all the bits to the right one place. That is to say, each bit moves over to a less significant place. The result is a divide by 2 in the same way that moving a decimal point to the left for a base-10 system is a divide by 10. A rotate left would be a multiply by 2 in the same way.

What happens to the last bit in the row? Assembly language has two kinds of rotates. The first, using *RR* or *RL*, is a simple 8-bit rotate where the left-most bit goes around to the right-most position or vice versa. *RRC* or *RLC*, on the other hand, includes the carry bit. If the carry is clear, then a zero rotates *into* the byte. If you rotate a multibyte number, the last bit of the first rotate moves *out* to the carry, ready to rotate *into* the *next* byte.

---

<sup>41</sup>For variety, this example uses the *CSEG* directive rather than naming the module and then making it relocatable. The difference relates to issues at linking time and is discussed in Chapter 8.

C supports only the *shift*. A shift is a rotate that *always* fills zeros on the incoming end and discards any bits falling off the other end. A shift of 8 bits on a *char* variable will always give zero.

**FAST MULTIPLY AND DIVIDE:** Depending on the cleverness of the C compiler, a shift can be faster than a multiply or divide. (Some compilers will actually substitute directly for a multiply by 2 or divide by 2, so it may not save anything). A shift will be somewhat obscure to a novice reader, too. However, when you involve fixed multiplication by 10, for example, it may be more efficient to shift left twice, add in the original, and shift left again. (Then again, if the compiler has already included the appropriate parts of the math library, there may be no code space savings and only minimal speed improvement.)

**ROTATE AND SHIFT; DON'T MODIFY VARIABLE:** Don't think a rotate or shift operator changes the variable shifted. Nothing changes unless you *assign* the result back to the original variable as in `x=x>>3;`.

## ASSIGNMENT OPERATORS

Unique to C is a shorthand representation for modifying a variable and reassigning the result back to the original variable. Normally you would write something like:

---

```
PORTA = PORTA & 0xf7;
```

---

to set the fourth bit up (bit 3 when you start counting with zero) low. In C, you can shorten this to:

---

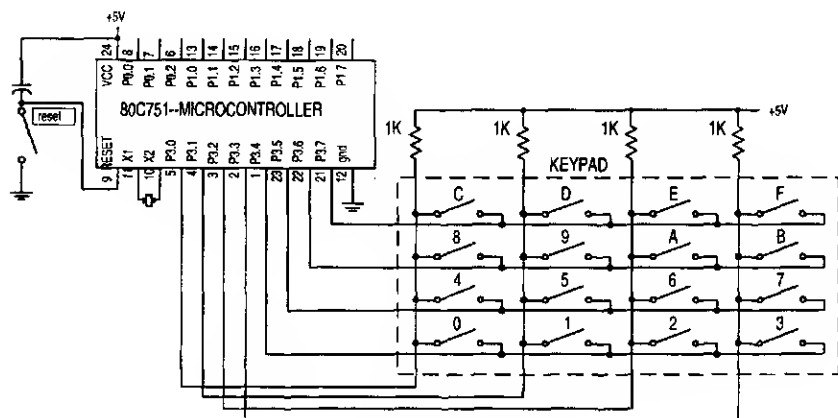
```
PORTA &= 0xf7;
```

---

using the assignment operator. The assignment operators are valid for +, -, \*, /, %, <<, >>, &, ^, and |.

## IDENTIFYING BIT CHANGES

Bitwise logic operators are useful to identify *changes*.<sup>42</sup> Assume you have a keypad for input. You can scan the matrix one column at a time. Each time you drive a column low, read in the rows. Pushed keys come in low, while the others keys in the driven column come in high.



**Keypad schematic**

If you repeat this column scanning every 40 msec, you will give prompt recognition of user inputs and still avoid switch bounce problems. The rest of the program needs only know of any *changes* to the inputs—new buttons pushed or old buttons released. With bitwise logical operators, you can easily sort out changes, as shown next.

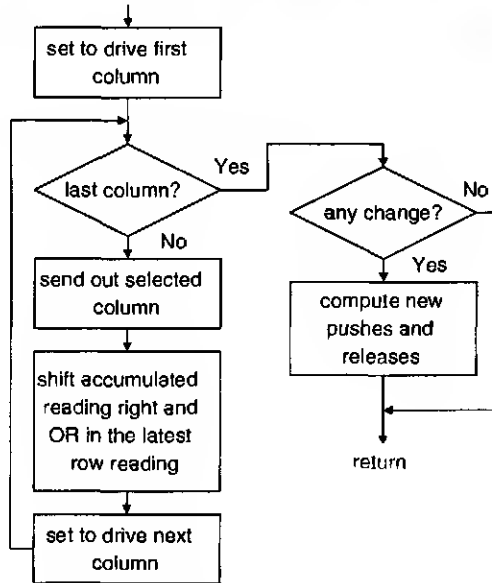
### Logic to Detect Key Changes

previous reading (old)	0 0 1 0 1 1 0 1
most recent reading (new)	0 1 0 0 1 1 0 0
old exclusive-OR new	0 1 1 0 0 0 0 1
new_pushes (old^new&new)	0 1 0 0 0 0 0 0
new_releases (old^new&old)	0 0 1 0 0 0 0 1

Here the bitwise logical operations easily and efficiently mark the changed keys.

<sup>42</sup>Suppose you decide to scan a matrix of keys directly rather than use an encoder chip. An encoder chip such as the 74922 is an alternative to scanning, saving three port bits but requiring one extra chip.

**NEGATIVE LOGIC:** It is easy to be confused by negative logic inputs. It is especially important to test any logic processing with a simple example—either by hand or with a program simulator.



**Keypad scanning flowchart**

The first part of the keyscan software is the scanning of a  $4 \times 4$  matrix. The high nibble (4 bits) of the port sends out the column drives, and the low nibble of the port reads back the rows. The program combines the read-back results (4 bits for each column driven) into a 2-byte number. The last step involves comparing the new reading with the previous reading to test for new pushes and new releases. The software was written for hardware that inverts, which makes the logic more intuitively pleasing by driving and reading back ones. How short the C program is! As you see it, though, it is not intuitively understandable.<sup>43</sup>

<sup>43</sup>The flowchart may help. Details of flowcharting are given in Chapter 5.

---

### Key Scan Program

```
#include <absacc.h>
44 #define PORTA XBYTE[0xffc0]
    unsigned old,new,push,rel,temp;
    unsigned char clmn_pat;
    void main(void){
45     for (clmn_pat=0x10;
           clmn_pat<>0;clmn_pat<<1){
        PORTA=PORTA & clmn_pat;
        new=(new<<4) | (PORTA & 0xf);
    }
46     if (temp=new ^ old)>0){
        push=temp & new;
        rel=temp & old;
    }
}
```

---

### KEY SCAN PROGRAM

```
PORTA EQU 6000H
DSEG
    OLD: DS 2
    NEW: DS 2
    PUSH: DS 2
    REL: DS 2
CSEG
47 SCAN:MOV DPTR,#PORTA
```

---

<sup>44</sup>This is an alternate way to define the external ports in Keil/Franklin C where you do not want to link in a separate assembly module. The header file includes the definition of the *XBYTE* function.

<sup>45</sup>This loop scans the keys. The *for* loop is discussed in the next chapter. By looking for the pattern (*clmn\_pat*) becoming zero, this will loop four times. After the fourth *shift left* (>>), the 1 bit will “fall off,” and the pattern will have all 0s. A *shift* fills 0s instead of bringing bits around as an assembly language rotate would do.

<sup>46</sup>This block identifies the *changes* since last time using the logic just described. C permits *embedded assignments*, so the *if* test includes the filling in of a value for *temp*. The use of *temp* could be avoided, but there might then be more code produced because the logical operation would have to be carried out three times. Notice the difference between the = here and the ==, which would only have done the test for equality without assigning any values.

<sup>47</sup>SCAN is a “complete” program but rather pointless since nothing is done with the results of the key scan. Chapter 7 goes into the details of subroutines, which would be a better use for this program. The *SHIFT* part is a *subroutine* since it ends with *RET* and is called from the main program..

```

48  MOV R0,#00010000B;COLUMN 1 LOOP:
    ANL R0,#11110000B ;4 BITS LOW
    MOV A,R0
    MOVX @DPTR,A ;DRIVE COLUMN
    MOVX A,@DPTR ;GET ROW READING
    MOV R2,#NEW ;STORE LOCATION
49  LCALL SHIFT ;STORE NIBBLE
    MOV A,R0
    RL A ;NEXT COLUMN-LEFT BY ONE
    MOV R0,A
    JNZ LOOP ;NOT DONE SCANNING
    MOV A,NEW ;TEST FOR CHANGES
    XRL A,OLD
    JNZ CHANGED
    MOV R0,A
    MOV A,NEW+1
    XRL A,OLD+1
    JZ DONE ;NO CHANGES-GO ON
    CHANGED:MOV R1,A
    MOV A,NEW ;GET NEW PUSHES
50  ANL A,R0
    MOV PUSH,A
    MOV A,NEW+1
    ANL A,R1
    MOV PUSH+1,A
    MOV A,OLD ;GET NEW RELEASES
    ANL A,R0
    MOV REL,A
    MOV A,OLD+1
    ANL A,R1

```

---

<sup>48</sup>The main program, *SCAN* involves going four times around *LOOP* in order each time to drive one column and read back the four rows. The result is put into the 2-byte storage called *NEW*.

<sup>49</sup>This routine is used to gather the 4-bit readings into a 16-bit (2-byte) result. Notice how the *RLC* allows the low-byte bits to carry across into the high-byte. When the previous value has moved over by four, then the lower 4 bits are masked and the newest 4-bits are ORed in.

<sup>50</sup>The logical operations described above to sort out new pushes and releases begin here.

```
MOV REL+1,A
DONE: MOV OLD,NEW ;UPDATE OLD
MOV OLD+1,NEW+1
LJMP SCAN ;START OVER

SHIFT: MOV R1,#4 ;LOW 4BITS => 16BITS
S2:CLR C
RLC @R2+1
RLC @R2
DJNZ R1,S2
ORL ACC,#0FH
ORL @R2,ACC
RET
END
```

---

## ARITHMETIC OPERATORS

Add, subtract, multiply, and divide are the math operations supported directly in the hardware of most microcontrollers. It is deceptive to say that the 8051 family *supports* all four math operations.<sup>51</sup> It supports them only for (unsigned) bytes. Instead, *groups* of assembly language instructions handle math for bigger variables. For *unsigned int* variables, C compilers will either code the necessary instructions in-line or add in the necessary function calls to a library function. *Complete* ANSI C compilers must support double precision, signed, and floating-point.

**ACCURACY:** Many operations in embedded control have very specific limits to the range and precision of numbers. Look very closely at what accuracy you *really* need. If the input is 8 bits, perhaps 8-bit math will suffice if you can be sure intermediate results won't overflow.

---

<sup>51</sup>Multiplication and division were not in the earliest machines such as the 8008 and 8080. You can get multiplication and division with successive addition or subtraction or through shift-and-add/subtract routines. Some 16- and 32-bit processors directly support unsigned integer math, but even there it has been common to rely on a math co-processor. There are at least two in the 8051 family that have an added co-processor on-chip for large variables, but even in those cases the co-processor is a separate hardware device rather than an extension to the instruction set.

**MATH VS. LOOKUP.** Especially with non-linear transformations (say the speed of a pump versus flow rate), it may be sufficient to use a lookup table (and linear interpolation if necessary) rather than the complex math arising out of some curve-fitting algorithm. Anything that can be pre-computed and put in a table will run much faster than a multibyte math operation.

If you are interested in writing your own math algorithms, an example in Chapter 8 (page 203) shows a set of assembly language double-precision math functions but most C programmers are quite content to rely on the libraries supplied with the compiler.

C automatically does *type conversion* (expanding bytes to word/integer, etc.). For example, if you add a byte to an integer, the result will be an integer. C has an operation to *force* a variable to a different type.

---

#### Automatic Type Conversion

```
unsigned int a,b;
unsigned char c;
52 a=b+c;
```

---



---

#### Casting

```
unsigned int a;
unsigned char b,c;
53 a=b+(unsigned int)c;
```

---

<sup>52</sup>The program takes *b*, treats *c* as though there were 8 high-order zeros attached, does the math, and assigns the result to the 2 bytes of *a*. If *a* were an *unsigned char*, then the 2-byte result of the math might have the upper byte discarded (even if that included non-zero bits).

<sup>53</sup>Where *b* and *c* are bytes and *a* is an *int*, the math might be carried out with single-byte precision, and the carry bit might be lost *before* the result is converted to a 2-byte quantity. Some C compilers for other processor families *promote* (change—enlarge) all *char* variables to integers. Here the parenthesis ahead of *C* casts it to an integer before the math is done so the intermediate result must be kept in an integer.



**TRUNCATING DUE TO VARIABLE SIZE:** By adding two 1-byte variables and assigning the result to a 2-byte variable without casting one of the 1-byte variables, the compiler may truncate the result to a byte at the addition and then promote it to 2 bytes when doing the assignment.

There are machine instructions that involve the carry bit, making it straightforward to process any number of bytes, one at a time, with the carry holding over between bytes. Because C handles 2-byte and larger variables in software (often from libraries) instead of by user programming, by looking at assembly language examples you can at least imagine what the general approach in the high-level libraries may be.

**DEDICATED MATH FUNCTIONS:** Sometimes running speed is at a premium, and the application needs only very specific math operations. Then it can be more efficient to write dedicated assembly subroutines to replace the more general library functions. For example, suppose a multiply is needed, but it will always be a multiply by 5. Or, an  $8 \times 16$  multiply is needed but you can guarantee in your application that the result will never exceed 16 bits. Since the library function may carry out a  $16 \times 16$  multiply and truncate the result, a shorter version which is less general may be more efficient. Chapter 8 on modular programming discusses more details of including custom functions.

The next example shows the addition or subtraction of two unsigned 16-bit numbers. Obviously, this is trivial in C.

---

#### 2-byte Addition and Subtraction

```
unsigned int x,y;  
x = x-y;  
x = x+y;
```

---

**CONFUSING REMAINDER WITH FRACTION.** Don't think the 8-bit divide with a 16-bit result leaves a whole number and a fraction. Actually, the divide leaves the remainder rather than a fraction.

---

#### 2-BYTE ADDITION

```
54 A16:MOV A,R3
      ADD A,R5
      MOV R3,A
      MOV A,R2
55   ADDC A,R4
      MOV A,R2
```

---

---

#### 2-BYTE SUBTRACTION

```
S16:MOV A,R3
56   CLR C
      SUBB A,R5
      MOV R3,A
      MOV A,R2
      SUBB A,R4
      MOV A,R2
```

---

<sup>54</sup>A problem with assembly language is the number of decisions about where to keep things. There are pointer-based instructions (based on *RO* or *R1*) and direct (to internal locations) instructions. Here, though, let's assume the first number is in *R2/R3* (high byte/low byte) and the second number (to be subtracted *from* the first in the second case) is in *R4/R5*. Suppose the result is to be overlaid on top of *R2/R3*. First add (or subtract) the two low bytes, being sure the carry bit does not come in at the bottom. In a more-than-two-byte routine, it might be better to clear the carry first and loop with the *ADDC* instruction.

<sup>55</sup>Be sure to include the carry bit without doing anything to destroy it in between. Add the second two bytes. If there is a carry when done, the result is more than 16 bits. What to do depends on what should happen next. If you need a fast, efficient routine, make sure that an overflow *can't* happen by checking all the possible values that could come to the routine in the final application. In more general routines, set an "error" bit variable, and have the calling routine check the flag before using the results.

<sup>56</sup>There is no instruction to subtract without including the carry bit, so it *must* be cleared first.

The next examples show an  $8 \times 8$  multiplication and an  $8 \times 8$  bit division. The only high-level language issues relate to ensuring you produce a 16-bit result.

---

#### One-byte Multiply

```
unsigned char x,y;
unsigned int z;
57 z = (unsigned int)x * y;
```

---



---

#### One-byte Divide

```
unsigned char x,y,a,b;
58 a = x / y;
59 b = x % y;
```

---



---

#### ONE-BYTE MULTIPLY

```
60 VAR SEGMENT DATA
61 RSEG VAR
    TERM1:DS 1
    TERM2:DS 1
MULT SEGMENT CODE
RSEG MULT
62 MOV R0,#TERM1
    MOV R1,#TERM2
```

---

<sup>57</sup>The *cast* is again used to ensure the production of a 16-bit result for the multiply. It may not be strictly necessary with some compilers, but emphasizes the fact that more than 8 bits are possible and that the result is generally put into the same type as the two initial variables. What would happen with mixtures of fixed and floating-point numbers is not certain, but it is not likely that the compiler would *automatically* convert them to fixed.

<sup>58</sup>The integer portion of the division is put into *a*. Remember that it is not a number with a fractional part.

<sup>59</sup>The % operator (called *mod* for modulus) obtains the *remainder*—not the fractional part. The treatment would be quite different for floating-point variables.

<sup>60</sup>In this assembly example, the multiply instruction exists directly, but the moves are of some interest. Assume *R0* points to the first (or numerator) term, and *R1* points to the second (or divisor) term. For variety, suppose you will leave the result (16 bits) in *R6/R7* in the usual high/low order.

<sup>61</sup>The variables are called *TERM1* and *TERM2* and are relocatable (assigned at link/locate time). If you want specific locations, define them with the *EQU* directive instead.

<sup>62</sup>Notice that *R0/R1* holds the *address* (not the value) of the variables. That way the pointer-based moves that follow access the correct values.

```

        MOV A,@R0
        MOV B,@R1
63      MUL AB
        MOV R7,A
64      MOV R6,B
        END

```

---

#### ONE-BYTE DIVIDE

```

VAR SEGMENT DATA
RSEG VAR
    TERM1:DS 1
    TERM2:DS 1
DIV SEGMENT CODE
RSEG DIV
    MOV R0,#TERM1
    MOV R1,#TERM2
    MOV A,@R0
    MOV B,@R1
65      DIV AB
    MOV R7,A
66      MOV R6,#0
        END

```

---

As an example of the mechanics of handling bigger numbers, the following function does 4-byte addition. The mechanics are the same as earlier, but a counter tells when the fourth byte is done. The values are on an artificial stack made up of *STKa* and *STKb*, with the msh of each at the lowest

---

<sup>63</sup>The multiply produces a 16-bit result with the high part in the *B* register. It also sets a flag (the carry) if the result exceeds one byte.

<sup>64</sup>Technically the instruction moves to *R6* from any direct internal address, and *B* is not an address. It may be the case that the direct address for the *B* register, *0F0H*, will need to replace *B* if your assembler is not clever enough.

<sup>65</sup>The quotient is left in *A* (the integer part).

<sup>66</sup>The integer remainder (not at all the same as the fractional result) is left in *B*, so if further division is to be done, it can follow. For example, 20 divided by 7 would leave 2 in *A* and 6 in *B*. You could then rotate *B* left by one (multiply by 2) to be 12, carry out the division by 7 again to fill the  $\frac{1}{2}$ -bit place, shift again to fill the  $\frac{1}{4}$ -bit place, etc. This approach is no more cumbersome than the shift-and-subtract type of division, but it might run slightly slower due to the time required for the *DIV* instruction.

address and the lsb at *STK+4*. It is a part of a family of functions discussed in the part of Chapter 8 discussing mixed languages. There you will also find functions to push this stack, take in an integer and extend it to 4 bytes, multiply, and divide.

---

#### Four-byte Addition

```

67 unsigned long stka, stkb;
    stka += stkb;

```

---



---

#### FOUR-BYTE ADDITION

```

ADDING SEGMENT CODE
STORE SEGMENT DATA
RSEG STORE
68 STKA: DS 4
    STKB: DS 4
    STKC: DS 4
RSEG ADDING
    DADD:MOV R0,#STKA+3 ;LSB OF A
        MOV R1,#STKB+3 ;LSB OF B
69    MOV R2,#4 ;4 BYTES TO PROCESS
        CLR C ;NO CARRY INTO FIRST ADDITION
    DAD1:MOV A,@R0
        ADDC A,@R1 ;A+B
70    MOV @R0,A ;SAVE IN A
        DEC R0 ;MOVE TO NEXT BYTE
        DEC R1
71    DJNZ R2,DAD1 ;4 TIMES

```

---

<sup>67</sup>Notice how simple the C version looks compared to the other language examples. That isn't to say it wouldn't produce a large amount of code, but the details are left to the compiler. Actually, the example used 30 bytes. If you need to, review the assignment operator on page 101 to convince yourself this is the same as *stka=stka+stkb*.

<sup>68</sup>Draw the values from *STKa* and *STKb* much like the previous example used *TERM1* and *TERM2*. The only registers available for pointers are *R0* and *R1*.

<sup>69</sup>It is more code efficient to use a counter rather than repeat the loop contents four times.

<sup>70</sup>The program decrements the pointers because the math must go from least to most significant, and the stack starts with msb at the lowest address.

<sup>71</sup>This powerful instruction decrements the counter and conditionally jumps in one instruction. When the addition of the fourth bytes is done, it goes on.

```

MOV R0, #STKC
MOV R1, #STKB
72 ACALL QMOV ;MOVE C OVER B
RET
QMOV:MOV R2, #4
73 QMO1:MOV A, @R0
MOV @R1, A
INC R0
INC R1
DJNZ R2, QMO1
RET
END

```

---

**FORGETTING OVERFLOW:** By omitting the cast when a result needs to be larger than either of the variables involved, it is possible the compiler will not promote the result until after the math is done. You may lose the upper part of the result.

At this point, it would be possible to go into multibyte math, including signed and floating variables. Since similar functions are available in C libraries and since *efficient* microcontroller programming tries to avoid such math, I will omit long and floating-point math.

## LOGICAL OPERATORS

One easily confused feature of C is the distinction between *bitwise* logical operators and *logical* operators. As mentioned already, bitwise logical operators modify the values of individual bits of a variable. **Logical** operators produce a

---

<sup>72</sup>Since  $A+B$  has been (arbitrarily) defined to destroy  $A$  and  $B$ , leaving the result in  $A$ , the last part moves the furthest (top? bottom?) term back one. If  $A$  or  $B$  is to be saved as in a stack-oriented math chip, it would first have to be duplicated on the stack. This is not the place to discuss stack-oriented math and *reverse Polish* calculations, but this approach is consistent with that approach.

<sup>73</sup>This subroutine is an example of breaking jobs up into pieces, as discussed in Chapter 7. Be careful when moving blocks or strings that the move doesn't overwrite the old values before they are all moved. For example, if the stack had four terms, then  $C$  should move down to  $B$  before  $D$  moves down to  $C$ .

*true* or *false* answer about the relationship between variables or expressions. Tests for loops and branches (the next chapter) use them extensively. The problem is that *bitwise* logical and *logical* operators have the *same* name in C. There are also both the *assignment equal* and the *logical test for equal*. The logical *equal* is `==` (a double equal) and the logical *OR* and *AND* are `&&` and `||`. Particularly problematic is the fact that the compiler can correctly accept either (with radically different results) in many expressions!

If a value of a variable in a logical test is greater than zero, it is *true*. A resulting shortcut is the omission of a *greater than zero* in a test for a bit: *if (PORTA | 0x40)...*; can suffice for: *if ((PORTA | 0x40)>0)....*.

## PRECEDENCE

How do compilers interpret lines with several operations? For example,  $A + B * C$  *could* mean, going left to right, to add *A* to *B* and *then* multiply *C* by the result. Actually though, it means to multiply *B* by *C* *before* adding *A*; as is normal in algebra, multiplication has higher *precedence* than addition. Using parentheses,  $A + (B * C)$ , avoids the problem and eliminates any doubt because you have been taught to evaluate the expressions nested within a `()` pair *before* using the result outside. In assembly, there is only one operation per line and a top-to-bottom flow, so there are no precedence issues. In C, the precedence rules are those of algebra, but mixed math and logic expressions can be misleading.

### IGNORING PRECEDENCE IN PORT MASKING

```
while (porta & 0x20 > 0) {}
```

Since the bitwise operator `&` has a *lower* precedence than the relational operator `>`, the first operation is really for testing whether `0x20` is greater than zero. The result is *always* true (which evaluates to a `0x1` or a `1`). Then you either do not mask the port at all, or you mask it for the lsb (bit 0) rather than bit 5. The final project using this error will mysteriously freeze or race on, depending on what happens with other bits or may totally ignore bit 5.

```
while ((porta & 0x20) > 0) {}
```

The correct alternative has the parentheses force the masking *before* the test for a non-zero result.

Here are all the precedence rules in tabular form. Later chapters will discuss some of the operators not already covered. Note the distinction between `!=`, `==`, and the other relational operators as well as between logical operators `&&` and bitwise logical operators `&`. If you write *understandable* code, these minor points of precedence will never come up because your liberal use of parentheses will avoid any confusion!

Precedence of operators

C operator (higher precedence toward top of table)	order of evaluation
( ) [ ] -> .	l to r
! ~ ++ * — (type) & sizeof	r to l
* / %	l to r
+ -	l to r
<< >>	l to r
< <= > >=	l to r
== !=	l to r
&	l to r
^	l to r
	l to r
&&	l to r
	l to r
? :	r to l
= += -= %=  = &=	r to l
,	l to r

## REVIEW AND BEYOND

1. Why are BASIC examples not included in this book?
2. Which variable types do the 8051-family processors directly support? Which ones require additional software functions?
3. What are the three main memory spaces in the 8051-family? How are the two types of RAM space different?
4. Are the line-feed and return characters significant in C source code (the program, as you write it)? Explain. Discuss indentation and features that might improve readability in a program listing.
5. What is a *type cast* in C?
6. What is an *assignment operator*?



7. Write a piece of program to compare a new 8-bit port reading with a previous value. Have the program produce a number having 1s only where the reading has 0s that were not present in the previous reading.
8. Explain some of the disadvantages of using floating-point math in C.
9. Which operators have the highest precedence in C?

# 5

## Looping and Branching

---

Having discussed *what* to do with numbers, the next step is to control *when* you do it. What is the use of a program that does one thing and stops forever? Microcontrollers are most useful when they take in outside information and use it to make decisions about what to do next.

### DECISIONS

If computers did only the basic operations described in Chapter 4, they would be of very little use. Their power lies in the ability to make decisions:

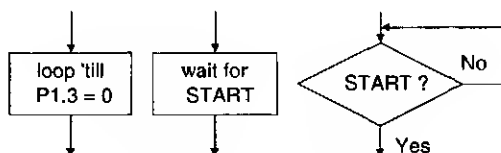
- Depending on the condition of a switch, either turn on the water or not.
- If this operation has run twenty-two times, go on and do the next operation.
- Keep on checking for the signal telling you that the speech chip can take in the code for the next word.

All these are examples of *decisions* that a microcontroller routinely makes. Based on the decision test, the program flow will *loop* (go back on itself) or *branch* (go in one of several possible directions).

### FLOWCHARTS

Flowcharts used throughout *this* book may be different from flowcharts you have seen before. The flowcharts here are *not* detailed pictures of the pro-

gram, but rather a *quick overview* of the method of solution. I argue that just as program code should be no longer than a page, so flowcharts should also fit on a single page. There are horror stories relating to many-page flowcharts with various paths to points several pages away. Although technically correct, such flowcharts do not help anyone understand what is going on in the program. Make *understandable* flowcharts. When they become too complex to fit on one page, simplify them (and the program the flowchart represents) by making subroutines. Expand the *details* of those subroutines in *separate* flowcharts on other pages. Thus, the *main* flowchart will always give an overview of the pieces of the program. If you need more detail as to the method of solution, go to the appropriate subroutine.



**Functional flowcharts**

All flowcharts should use functional names and should generally *not* refer to specific variable names. For example, consider the function to wait for someone to push a button, shown here. It is better to flowchart *wait for START button*, rather than *loop until P1.3=0*. Better yet might be a decision block. The rest of the chapter includes examples of suggested flowchart symbols.<sup>1</sup>

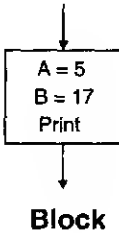
## STRUCTURED LANGUAGE

C is a *structured language*—there are rigid rules that prohibit crisscrossed program flow. A structured language never allows jumps into or out of a function without saving or restoring the stack and any other pertinent regis-

---

<sup>1</sup>Computer science, particularly in the area of data processing, has a much more elaborate set of symbols. Most additional symbols relate to complex subsystems as parts of the program flow and are not appropriate here. There are several other methods of showing iterative loops in flowcharts, which may be more efficient, but I will not cover them here.

ters. With structured programming, apart from the special case of interrupts discussed later, you cannot corrupt the stack with any acceptable commands.<sup>2</sup>



The basic element of structured languages is the *block*. It is a piece of program where the flow enters at *only* one place and leaves at *only* one place. There can be no “sneaking” into the middle sometimes or leaving partway through the block. The example shows a simple block structure in C. The three program examples here show forms of the same block.<sup>3</sup>

---

#### Basic Block Structure

```
{
    a=5;
    b=17;
    print;
}
```

---



---

#### Alternate Styles in a Block

```
{
    A=5; B=17; print();
}
```

---

```
{A=5; B=17; print();}
```

---

The differences relate to the fact that C pays no attention to line breaks. The individual C statements do not have to be on separate lines. You can make program listings shorter and wider that way. This can be useful where several short expressions have a closely related function.

---

<sup>2</sup>You *can* follow structured programming rules even with assembly and BASIC—there is even a language called *Structured BASIC*.

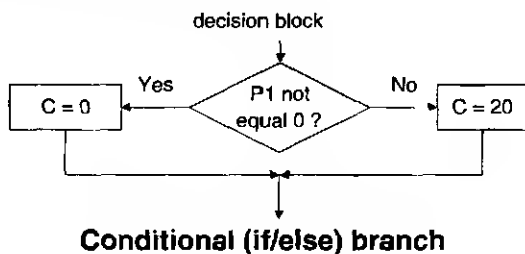
<sup>3</sup>Assembly does not define a block per se. You can think of a straight-line piece of program with no jumps or labels as a block in that it has only one entry and one exit point.

## BRANCHING CONSTRUCTS

The pages that follow discuss branching and looping constructs. The assembly language examples show a series of instructions to accomplish the same function as the high-level languages.

### If/else

A basic decision or branch in program flow is the *if/else* block. The else is optional, and the expressions can be blocks in themselves. In C, the *if/else* structure is quite straightforward.




---

#### If/else decision

```
if (P1!=0) c=20;
else c=0;
```

---

In assembly, use the *JZ*, *JNZ*, *JB*, *JNB*, *JC*, and *JNC* instructions. All the conditional branching involves jumps—there are no conditional calls available with the 8051. There are several *unusual* looping assembly instructions discussed in the following pages. With the carry bit, be careful an instruction between the carry setting place and the actual jump test does not change its value.

---

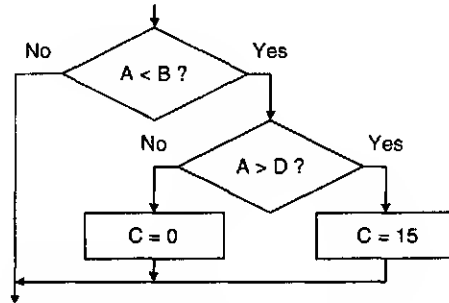
#### IF/ELSE DECISION

```
MOV R0, #C
MOV A, P1
4 JZ X
Y: MOV @R0, #20H
   SJMP Z
X: MOV @R0, #0
Z:
```

---

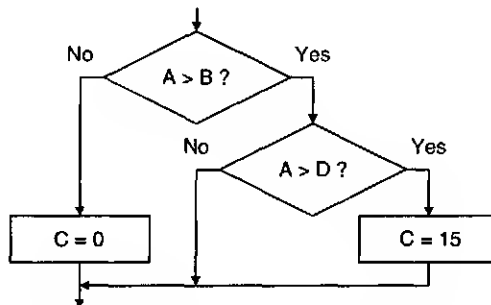
<sup>4</sup>The *JZ/JNZ* instruction depends on the accumulator at that moment—there is no zero flag as with the 8080 family

C allows assignments *within* the test expression, called *embedded assignments*. This is where the `++i` as opposed to the `i++` can be significant—is the variable incremented *before* or *after* doing the test? If you are only casually reviewing a program you can easily miss that sort of distinction.



**Nested if/else—versions 1 and 2**

The *else* blocks can be *nested*, as shown next. The nesting of the *ifs* ties the *else* with the most recent *if* unless the block structure defines it otherwise. The first and second code pieces in the next example are identical, though the (misleading) indentation suggests that the second code piece will set `C = 0` if `A` is not greater than `B`. If that is what you want, then, in the third code piece, the necessary `{ }` block shows how to make the *else* apply to the first *if*.



**If/else flowchart—third version**

---

**Nested if/else Blocks—versions 1, 2, and 3**

```

if (a>b){
    if (a>d) c = 15;
    else c = 0;
}
  
```

---

```

if (a>b)
    if (a>d) c = 15;
    else c = 0;

```

---

```

if (a>b) {
    if (a>d) c = 15;
}
else c = 0;

```

---

**FOOLING YOURSELF WITH INDENTATION:** Don't assume the *if* applies to several lines when it applies to only the first line. Unless *{}* surrounds the group, the rest of the lines will be part of the resumed flow.

```

if (a>b)
    c=25;
    g=a+b;
    p1=7;

```

If you are careful this will never be a problem. If not, many forms of this error will cause compiler errors, but fixing the errors by inserting *{}* until the errors "go away" may give the wrong result.

## Conditional Operator

An expression unique to C is the conditional operator. It is shorthand for an *if/else* decision where the two choices simply assign a different value to a variable. It is a test where the true condition assigns the first value and the false condition assigns the second value.

---

### Conditional operator

```
5c = (a>d)?15:0;
```

---

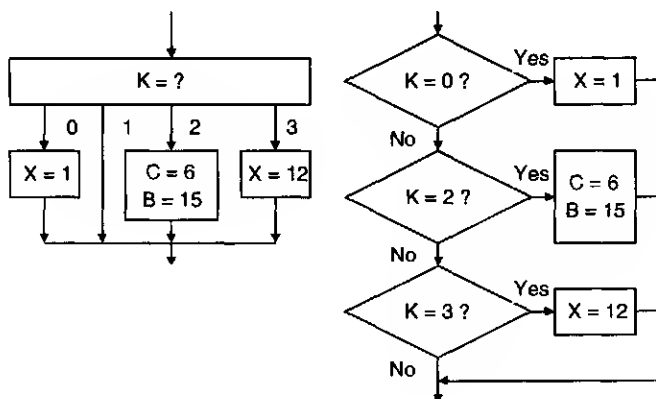
## Switch

Besides the simple branching construct of the *if/else*, there is in C the *switch* construct. It allows for branching out many ways based on the value of an expression. A string of *if/else* constructs can do the same job, but a *switch* can make a multiway branch more understandable. The simplest

---

<sup>5</sup>If *a* is greater than *d* then *c* equals 15; otherwise *c* equals 0.

form is when the branching depends on an integer, but it can also depend on the value of an expression. You do not have to represent all the possible cases—you may have a default case, or any non-existent cases will just fall through.



Case branch

---

#### Basic case branch

```

switch (k){
6,7  case 0:
        x=1;
        break;
        case 2:
        c=6;
        b=15;
        break;
        case 3: x=12;
8      default: break;
}

```

---

<sup>6</sup>Each case is tested against the *constant* value following the case, but it is not necessary to have them in any particular order. Any missing cases will cause no action.

<sup>7</sup>Program flow will *fall through* to the next case unless there is a *break*. In other words, without the break, case 0 would execute cases 2 and 3 as well. It is unnecessary to put braces around the cases since by default execution goes until you encounter the break.

<sup>8</sup>The *default* (if *none* of the listed cases match, then do the default) is optional. In any case, all flow resumes after the *}*, and values of *k* except the ones specified in the other cases will safely fall out as well. The default in this example is totally unnecessary since a case not in the list will automatically exit—it makes more sense if there is some alternate processing for *none of the above*.



**FORGETTING THE BREAK IN C:** Without the break the program will flow down through the following cases. You might assume each case would terminate when done and resume after the switch block, but the streams of flow for successive cases will add together into a river if you omit the break. This can be powerful for some applications, but it is a nuisance in most cases.

### Breaking Out: The goto

C does provide the *goto* statement somewhat corresponding to an assembly language jump.<sup>9</sup> The only allowed *goto* is to a labeled line within the same function. It is not permissible to do a *goto* out of a procedure into another procedure.

Probably a *goto* is *never* necessary. The only commonly accepted use relates to leaving a loop when errors come at different levels of nesting. *gotos* can also cut short a search when you reach the goal. You can do both actions without the *goto*, but it is sometimes more understandable with the *goto* included.

C has an additional set of seldom-discussed instructions for ending loops—the *break* and *continue* directives. The former is used with the *switch* construct and the latter causes flow to drop out of a *while* loop.

## LOOPING CONSTRUCTS

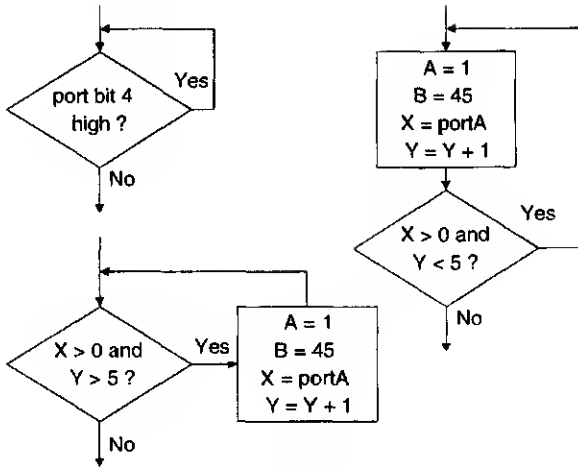
The branching constructs all carry the flow *forward* (except for some uses of *goto*). With *looping* constructs it is possible to *repeat* things.

### While loop

One looping construct is the *while* block. Program flow continues in the loop until the test fails. One form uses the test *first*, entering the block *only* if the test passes. If not, then the flow skips over the block and continues with the first statement *after* the block. A second form, the *do ... while()* loop, does the test at the *end* of the block to decide whether to go back and do it again or to continue. Thus, the block always executes at least once.<sup>10</sup>

<sup>9</sup>Of course, machine code jumps *result* from structured programming, but the structure keeps the stack depth correct and restores variables. If in assembly you *jump* into the middle of a routine and subsequently leave by a *return*, you will pop something other than the calling address and will return to some unplanned address.

<sup>10</sup>The two forms correspond to *DO WHILE* and *DO UNTIL* in Pascal, which may be clearer.



### While loops

---

#### Simple (empty) while loop

```
11 while ((P1 & 0x10)==0);
```

---

#### Normal while loop

```
while (x>0 && y++==5){
    a=1;
    b=45;
    x=P1;
}
```

---

#### do while loop

```
do{
12     a=1;
13     b=45;
    x=P1;
}
while (x>0 && y++==5);
```

---

<sup>11</sup>By testing *until* the fifth bit (bit 4) goes *high*, this loop waits for some signal from a user or external hardware. It is safer to test with a greater than zero rather than an equal to 10H because the latter leaves the possibility that the masking number and the equality test number might get messed up and not agree.

<sup>12</sup>The incrementing of the value of *y* is done by the *y++*. Remember that the *==* is different from the *=* which would *change* the value of *y*.

<sup>13</sup>This block will execute the block once before doing any testing of the values of *x* and *y*.

One of the unusual and powerful assembly instructions is for carrying out looping. The *CJNE* instruction compares the first two operands and branches only if they are not equal. It can easily make a loop to test a port until a particular value appears.

---

#### WHILE LOOP

```

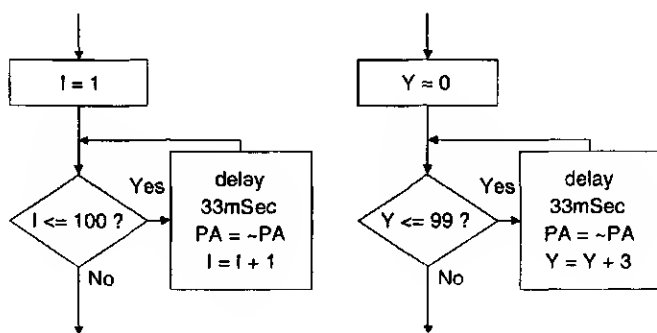
MOV DPTR, #PORTA
X:  MOVX A, @DPTR
    ANL A, 00010000B
14  CJNE A, 00010000B, X

```

---

### Iterative loop

A second very common C structure is the *iterative* loop. It uses constants, variables, or even complex expressions to control the number of times around the loop. There are three parts to the instruction. First is the *initial expression*. This usually assigns a numeric value, but it can execute *any* expression when you first enter the loop. Then there is the *test for ending* the loop. This usually tests for an upper value of an index, but you can have *any* test whose failure will end the loop. Finally, there is the *index increment*. Usually it is positive. Each time around the loop you increment (or decrement) the index variable. Again this can hold *any* operation or expression to be performed after failing the exit test before re-entering the block. You can do nearly incomprehensible things with such a construct if you are feeling powerful and secretive!



Iterative loops

---

<sup>14</sup>By testing until the fifth bit (bit 4) goes high, this loop waits for some signal from a user or external hardware. The *CJNE* is very useful here.

---

### Iterative Loop Structures

```

for (i=1;i<=100;i++){
    delay(33);
    px=~px;
}

for (y=0;y<=99;y=y+3){
    delay(33);
15    px=y;
}

for (da=start;status==busy;leds=~leds){
16    delay(33);
}

```

---

Assembly does not have the power of C in a single line, but one instruction, *DJNZ*, is quite powerful for small iterative loops. It can use one of the registers or a specified internal RAM location as the index counter. By its nature, it *decrements by one* each time, but for other decrements, put additional instructions just before the loop test.

---

### ITERATIVE LOOP STRUCTURE

```

MOV R0, #100
X:MOV R1, #33
    LCALL DELAY
    MOV A, P1
    CPL A
    MOVX P1, A
    DJNZ R0, X

```

---

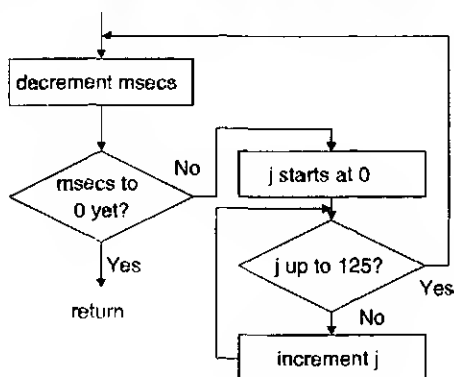


---

<sup>15</sup>This example increments *y* by three each time around the loop. The *i++* could be put into the second part of the *for* (*::*) expression, and the third part could be left blank! You could even put the body of the operation into the third part! Such embedding can be quite powerful, but also quite confusing.

<sup>16</sup>This example would use *#define* to designate the various port and bit assignments. It would start an A-D converter and wait for it to complete the conversion. It would toggle some LEDs at a rate determined by the delay and appear to the casual reader to do nothing but delay in the loop. For an A-D converter, the next action after the loop ought to be to pick up the completed reading.

**MIDDLE TEST OF FOR LOOP** In C it is easy to confuse the middle term of the loop, which must be *true* to continue in the loop, with a term which is the *ending* point of a counter.



**Software delay flowchart**

### EXAMPLE: TIME DELAY

It is common to produce a time delay by nesting loops so that the instruction execution consumes a known amount of time. The next example shows such time delays.<sup>17</sup> The time in the loop depends on the number of clock cycles and the crystal frequency. The delay function here consumes about one msec for each unit passed to it. By passing a value of 50 you get a delay of about  $50 \times 100 = 5000 \mu\text{sec} = 5 \text{ msec}$ .

#### Delay by Software Looping

```

18 void msec(unsigned int x){
    unsigned char j;
    while (x-- > 0){
19         for (j=0; j<125; j++){;}
    }
}
  
```

<sup>17</sup>This continues the delay discussion begun on pages 22 and 23. Chapter 7 discusses in detail routines and passing parameters—the focus here is on the loops involved. (Chapter 11 introduces getting a time delay from the use of the internal timer and interrupt.)

<sup>18</sup>This routine can take integer values (rather than bytes) to produce long delays. It is not precise as written although it is approximately correct based on an analysis of the assembly showing the most inner loop with *j* taking 8  $\mu\text{sec}$ . Different compilers may come out with quite different times and you could adjust the value of 125 empirically to compensate. Delay loops are often an inefficient way to get delays. For example, interrupts will stretch out time for software delays since the loop will stop incrementing during the interrupt routine.

<sup>19</sup>The null statement reminds the reader that it is intentionally empty. Actually the braces are unnecessary—you can have just the ;.

---

**DELAY BY SOFTWARE LOOPING**

```
20 PUBLIC MSEC
   MSEC SEGMENT CODE
   RSEG MSEC
      MSEC:JZ X ;QUIT IF A = 0
21      MOV R0,#250 ;250 X 4 = 1000
22      Z:NOP
          NOP
          DJNZ R0,Z ;4 USEC PER LOOP
          DJNZ ACC,MSEC
      X:RET
END
```

---

**REVIEW AND BEYOND**

1. Show how a program can exit a *for* block early without using a *goto*.
2. What are some possible pitfalls of unstructured languages?
3. Some argue that the *goto* is unnecessary in a structured language. What situations would possibly be easier to handle with a *goto* than with other structures?
4. What are the restrictions on use of *goto*?
5. What is the difference between a *while* and a *do...while*?
6. What happens in C if you omit the *break* in a *switch* operation?

---

<sup>20</sup>This module is made public so it could be linked with a program needing a time delay as will be discussed in Chapter 8.

<sup>21</sup>The delay could be fine-tuned to compensate for the overhead at the start and the testing the accumulator for the end. Technically, the delay is still not precise since there are 4  $\mu$ sec lost for the *LCALL* and *RET*.

<sup>22</sup>The *NOPs* bring the inner loop up to 4  $\mu$ sec so the index (*R0*) can fit in a byte for a 1 msec delay. Note that this routine destroys the value in *R0*.

# 6

## Arrays and Pointers

---

Up to now the programming has kept each variable in a specific place—either of your choosing or of the linker/locator's choice. If you are going to have one piece of code operate on different variables at different times, you need the ability to *point* to one variable now and later, point to another variable. One of the strengths of C is the ability to reference variables by pointers. The variable is *based on* the pointer—it is a *based* variable. This chapter starts with arrays, goes on to structures, and then plunges into pointers, based arrays, based structures, unions, and a few of the many obscure combinations of the whole lot. For *Just-in-time* learning, only study as far in as you see a need now, and just skim the rest for future reference. You can learn these types best with a specific need at hand.

### ARRAYS

An array is a collection of variables referenced by a common name.<sup>1</sup> There is a close relationship between arrays and pointers, but you can use arrays in C without understanding pointers. Arrays involve storage in a block of *connected* memory. For a *byte* (*char*) array, the bytes occupy successive memory locations.<sup>2</sup>

---

<sup>1</sup>They must all be the same type—for example, all *unsigned char* or all *int*. Otherwise you have a structure, covered later in this chapter.

<sup>2</sup>For an *int* array, they occupy successive byte pairs. *Long* and *float* arrays are groups of 4 bytes, and, depending on the compiler, *pointer* arrays have 1-, 2- or 3-byte groups. Useful as it might be for graphics, I have never heard of support in C for *bit* arrays.

---

### Accessing an Array Element

```
unsigned int ary[20];
unsigned int x;
ary[9] = x;
```

---

In assembly, you access an array by storing a *start address* for the group of bytes and then adding to it the value of an *index*. Supposing you want the fourth element of a byte array: Fetch the *start address* and then add 3 to it. Incidentally, that is why C begins counting array elements with zero—it directly adds the index to the start address. The first element, having index of zero, is the start address with a zero added. Unlike humans, computers prefer to start counting with zero rather than one.

In assembly language, for arrays using *internal* memory, there are direct access instructions using *R0* and *R1* as pointers. Because of limited space, however, it is more common to keep arrays in off-chip RAM or ROM. The example that follows shows a subroutine that returns the *R2<sup>th</sup>* element of an array in off-chip RAM whose starting address was loaded into *DPTR*. It adds the value in *R2* to the pointer to set the pointer to the correct place in the array.

---

### ACCESSING AN ARRAY ELEMENT

```
FETCH:MOV A,R2 ;GET THE INDEX
      ADD A,DPL ;DPL IS AT 82H
      MOV DPL,A
      CLR A
      ADDC A,DPH ;DPH IS AT 83H
      MOV DPH,A
      MOVX A,@DPTR ;GET ARRAY VALUE INTO A
      RET
```

---

**STARTING ARRAYS WITH 1 RATHER THAN 0:** The "one<sup>st</sup>" element is the second byte.

**LETTING THE ARRAY INDEX REACH THE ARRAY SIZE:** The index for a 20-byte array should never go above 19 since that is the top element. There is no error checking—if you overrun the array, the variable stored at the next higher address will unintentionally be involved.



Arrays are much easier to visualize in high-level languages. The next example sets up an array, *ary* that has 20 2-byte (*unsigned int*) members and then copies *x* into the tenth place. The compiler actually assigns *x* to the eighteenth and nineteenth locations above the start of the array. (The 0 element into bytes 0 and 1, the first element into bytes 2 and 3 . . . the ninth element into bytes 18 and 19).

**LARGE ARRAYS:** Be careful to size arrays only for what is needed. When most of an array is not used—especially with a multidimensional array—you can tie up large amounts of memory. Embedded controller software, unlike large multiuse systems where the memory blocks are large and are “there anyway,” should not require unnecessary RAM. Memory is not just sitting around free for use in an embedded system; if called for in the software, it will have to be built into the system.

In C, there can be two-dimensional arrays.<sup>3</sup> The next example shows a two-dimensional array of floating-point variables. It retrieves a value and puts it into another floating-point variable.<sup>4</sup> In this case (*[5][0]*) the compiler points to the twentieth byte from the start of the array (*C8*) because the first of the “rows” of ten elements each take up 20 bytes. The second example shows array initialization for a series of message strings.

---

#### Two-dimensional Arrays

```
float xdata ary2d [10][10];  
float xdata x;  
x = ary2d[5][0];
```

---

<sup>3</sup>The ANSI standard (but not all C compilers) even supports more than two dimensions, but the use is quite limited in microcontrollers because you can easily run out of memory that way. For example, a  $10 \times 10 \times 10$  array of *float* values requires almost 4k bytes. A  $25 \times 25 \times 25$  array exceeds the entire possible 64k byte xdata space.

<sup>4</sup>After the following discussion of pointers, I will discuss a second way to access arrays in C, but this is the more obvious way.

```

5  uchar code msg[][17]=
6  {"This is a test",\n},
   {"message 1",\n},
   {"message 2",\n});

```

---

## LOOKUP TABLES

The use of arrays is well suited to lookup tables as is shown next. In many embedded applications it is more efficient to use tables rather than mathematical computations because you can execute a lookup quickly and it usually involves less code than a math algorithm. Compute the tables in advance and include them in ROM space. Many science-oriented users, fed a steady diet of equations, expect a controller to carry out the math, on demand, to a precision that is unnecessary for most applications.

---

### Temperature Conversion by Lookup

```

#define uchar unsigned char
7  uchar code CFTbl[] = {32,34,36,37,39,41};
8  uchar F,C;
   uchar CtoF(uchar degc){
9     return CFTbl[degc];
   }

```

---

<sup>5</sup>This is a two-dimensional array. You *must* enter the second size since it is not determined from the list. The first dimension (3) is determined from the number of inner curly brace pairs.

<sup>6</sup>Two-dimensional arrays need *two* sets of curly braces for initialization.

<sup>7</sup>In this case, the arrays are set up to be in the *code* (ROM) space and are made *unsized* in the sense that the size is determined by the number of entries in the list. That helps avoid the effort and error possibility of counting the list to see how many there are. Also, the *define* of *uchar* is much simpler to type and allows the attention to focus on the variable rather than on the word *unsigned*. The size is fixed at compile time—it does not change dynamically while running as it might with BASIC.

<sup>8</sup>Although the example specifies that the array be located in *code* or *xdata* space, this is not *standard C*. There, the initialization of an array would involve an entire series of program instructions *automatically* generated to move the code to RAM space at startup. Standard C requires that someone sets all variables not otherwise initialized to zero. In embedded 8051 systems that is a waste of time and code space! Good compilers should make it possible to suppress that unnecessary and wasteful initialization.

<sup>9</sup>The returning of a parameter is discussed in Chapter 7 where functions and procedures are formally introduced.

```

main() {
    C=5;
    F = CtoF(C);
}

```

---

#### TEMPERATURE CONVERSION BY LOOKUP

```

;WORKS FOR 0 TO 5 DEGREES C INPUT
;PUT C VALUE IN ACC BEFORE CALLING
CTOF:MOV DPTR,#TEMPTBL ;POINT TO TABLE
10   MOVC A,@A+DPTR ;GET IT FROM TABLE
    RET
TEMPTBL: DB 32,34,36,37,39,41

```

---

**CONSTANT ARRAYS:** It is often much quicker to look up values in a table than it is to compute them. The 8051 has an instruction specifically for fetching values from a table in ROM. Even for precision above 8 bits it is often sufficient to linearly interpolate between table values and can involve less code space and time than a complex floating-point algorithm.

## STRUCTURES

A *structure* is a group of related variables referenced by one name. For assembly language, you handle structures only indirectly as individual variables or by special handling of an index. In C, however, structures can be very useful. It may help to think of a 2-byte variable as a structure made up of 2 bytes. The first byte is the high part of the binary number, and the second byte is the low part. When you make an array of 2-byte values, each one consists of a high-low byte pair. Together they make up the entire number.

---

<sup>10</sup>In assembly, there is a handy instruction for accessing arrays in ROM space (often used for fast lookup conversion tables), the *MOVC* command. Since the instruction adds *A* to *DPTR*, a subroutine to convert degrees Celsius to degrees Fahrenheit could work quite simply. Since the *MOVC* instruction includes the addition step, the operation is quite efficient. Many years ago, the initial developers of the 8051 obviously had this sort of application in mind.

On a larger scale, a structure could represent the information about the solenoids of a sequencer. One part could be the 32 bits that represent the 32 solenoids. A 1 for a bit could mean that a solenoid is "on." Then another part of the structure could be the amount of time to maintain the state, and a third part could represent whether the particular state is the last in the sequence. The example shows such a structure.<sup>11</sup>

---

#### Structure Example

```
struct{unsigned long s;  
12     unsigned int t;  
        unsigned char done;} state;  
13 state.t = 321; /*use of structure */
```

---

**CONFUSING A STRUCTURE ELEMENT WITH A VARIABLE BY THE SAME NAME:** Forgetting the overall structure name and dot when referencing a structure element may produce a compiler error if there happens to be another variable with the same name as the structure element.

**PUTTING BITS IN A STRUCTURE:** Since bits can only be in one part of on-chip RAM, an error will result if you define a structure made up of both bits and other data types. A structure has to involve contiguous bytes.

If you have several different structures with the same form you can define the form separately in a *structure template*. The name for the form is the *structure tag*, shown here for the previous example. You could use the tag repeatedly as new structures are declared.

---

<sup>11</sup>The assembly shown is the output from the compiler. In human-generated assembly the treatment is up to the programmer to include with index pointers and starting addresses to access a (contiguous) block of bytes.

<sup>12</sup>This sets up a single structure named *state*, which contains 7 bytes.

<sup>13</sup>This line shows assigning a value to an element of the structure named *state*. The *period* or *dot* (.) is used to specify an element within the overall structure. The first 4 bytes are the 32 on/off bits for the solenoids, while the next 2 bits hold the 16-bit number which is the time to the next state transition. Finally, the last byte holds the 1 or 0 that indicates whether to go on to another state or start over.

---

### Structure Templates

```
#define uchar unsigned char
#define uint unsigned
struct stateform {
    unsigned long s;
    uint t;
    uchar done;
};
struct stateform state;
state.t=321 /*use of structure*/
```

---

## NEW DATA TYPES: TYPEDEF

This is much like a *#define* statement where the name substitutes for the variable declaration. The example below shows a *typedef* for coordinate pairs setting up an array of pairs and accessing them.

---

### Defining a New Type (*coord*)

```
#define uchar unsigned char
uchar w;
typedef struct{uchar x,y;}coord;
14 coord move[20];
void main(void){
    w=move[3].x;
}
```

---

The main advantage of a *typedef* is the moving of the variable definitions to a single line. If you decided later that you must keep all the coordinate data in *unsigned int* variables, you can just change the information in the *typedef* line. It also helps to make the purpose of a structure more obvious.

## ARRAY OF STRUCTURES

In many cases, it would make more sense to use an *array of structures*. Taking the previous structure and making an array is quite direct, as shown on the next page.

---

<sup>14</sup>Here I declare an array of twenty variables of type *coord*. Actually, it is an array of structures—here holding 2 bytes each.

---

### Array of Structures

```

#define uchar unsigned char
#define uint unsigned
struct stateform {unsigned long s;
                  uint t; uchar done;};
15 struct stateform state [20];
16 state[11].s |= 0x04000000;

```

---

## ARRAYS WITHIN STRUCTURES

Finally, in C it is possible to have *arrays within structures*. In the previous example it might well be more efficient in the final machine code to avoid reference to the larger variable types which, depending on the compiler, might bring in large, undesired libraries. The previous example becomes that shown here.

---

### Array of Structures with Arrays

```

#define uchar unsigned char
#define uint unsigned
struct stateform {uchar s[4];uint t;
                  uchar done;};
    struct stateform state [20];
17 state[11].s[0] |= 0x04;

```

---

The C language can go on from here with *nested structures*. It is a challenge to come up with a real embedded control example, but I suppose you could take the previous example and add a separate piece of information related to incoming sensor pulses for repair logging, as shown on the next page.

---

<sup>15</sup>Either the large memory model must be used or else the *xdata* keyword should come ahead of *struct* since this structure won't fit in on-chip RAM.

<sup>16</sup>The array is accessed with the [ ] as usual. The dot indicates that the particular element within the structure is coming next. The 32 *bits* are all part of one *long* integer. The example involves 140 (20 × 7) bytes of storage. The hex representation looks long because it addresses the *entire* 32 bits. You should avoid decimal notation since it obscures the fact that variable is not being used as a number. Using the |= operator saves space by not having to repeat the structure reference.

<sup>17</sup>This is the seventy-eighth byte of the structure.

---

### Nested Structures

```
#define uchar unsigned char
#define uint unsigned
struct sf{uchar s[4];uint t;uchar dne;};
struct repairtype{struct sf state;
                  uchar sensorcount;};
struct repairtype repairs [20];
18 repairs[11].state.s[0] |= 0x04;
```

---

### Nested Structures (Without Tags)

```
#define uchar unsigned char
#define uint unsigned
19 struct{struct{uchar s[4];uint t;
              uchar dne;}state;uchar
              sensorcount;}repairs[20];
repairs[11].state.s[0] |= 0x04;
```

---

```
struct{struct{uchar s[4];uint t;
              uchar dne;} state; uchar
              sensorcount;} repairs [20];
repairs[11].state.s[0] |= 0x04;
```

---

This example is not the height of simplicity and is probably unnecessary for embedded applications. It does see use for data processing where a structure could hold an employee's name and address and other structures could use that structure template with additional parts for other information. If you *really* want to lose the casual reader, nest this all in one expression without tags, as shown in the last version above! In addition, put it all on as few lines as possible. It will not bother the compiler, which ignores line feeds anyway, but it will certainly amaze your friends!

## CHOOSING MEMORY SPACES FOR VARIABLES

With the 8051 family of controllers, there are at least three different *types* of memory. We have been talking about variables by their *name*, ignoring the

---

<sup>18</sup>This is the eighty-ninth byte of the structure (I didn't count but relied on the results of a simulator to tell this!).

<sup>19</sup>The best indentation style for this is unclear. I would tend toward packing more on each line, but sometimes it is desirable to fit comments in with each element of the structure.

details of *where* we store them.<sup>20</sup> In assembly you make the choice quite deliberately—there are only a few instructions (*MOVX* and *MOVC*) that address off-chip memory.

C prefers to leave the assignment of specific locations up to the compiler unless you *purposely* override them.<sup>21</sup> In C, you can leave the actual memory space decision to the *memory model*, which defaults to *small* (all variables in on-chip RAM). If *small* is used, the compiler will store every variable *not otherwise specified* in on-chip memory. For small programs with few variables, there is enough on-chip memory space.

Larger memory models assign variables to off-chip memory.<sup>22</sup> You can choose memory-space for each variable individually as you define it using extensions like *xdata*, *code*, or *data*.

**UNINITIALIZED POINTER:** If you use a pointer-based variable before you set the pointer to a specific place the pointer will point randomly (or point to zero if the code to initialize the variables has been included by the linker). The effect could be bad in either case. If the pointer is *uninitialized* it is like randomly flipping a loaded gun. If you pull the trigger you might hit anything!

## POINTERS

A *pointer* is a *variable* that holds the *address* of another variable. The variable to which a pointer points is a *based variable*. In assembly, you could use a byte put into *R0* or *R1* to point to internal memory space, or else a 2-byte value put into *DPTR* to point to external RAM or code (EPROM) space. C pointers are more complicated. Here is a C example of setting up a pointer to a variable and then using the variable pointed to.

<sup>20</sup>Even in assembly, the use of the DS directive and relocatable segments can allow the exact location where you keep the value of a variable to be determined at the time the pieces are linked. You have to define the *type* of segment.

<sup>21</sup>You need specific addresses for I/O devices; there is usually no prewritten driver to insulate you from such details. Port chips, A-D chips, and other added I/O have absolute addresses determined by the wiring and address decoding of *your* design. If the compiler/linker system chose to put the I/O at some other address, it would cause disaster.

<sup>22</sup>If you get beyond the 64K byte limit, *bank-switched* memory schemes are an option discussed later. Also when running out of room, the *stack-oriented* approach can apply. Because the 8051's hardware stack is in internal on-chip RAM, which is very small, and because *standard* C is highly stack-oriented, some compilers make a stack in off-chip RAM using software instructions. The call return addresses are put on the internal stack, but all the other passed parameters and perhaps some temporary results go on this artificial stack.



---

### Based/pointer Variable

```

#define uchar unsigned char
uchar count;
23 uchar *x;
24 uchar xdata *y;
    uchar data *z;
    uchar code *w;
    uchar data *xdata zz;
25 x = &count;
26 *x = 0xfe;

```

---

**ADDRESSING THE WRONG MEMORY SPACE:** By mixing up *code* and *xdata* pointers, when you send a message to a display function, if the message is a "canned" one in EPROM, it could be brought from *xdata* space (the *RD* line on the 8051) rather than *code* space (the *PSEN* line on the 8051). If the display function works out of *code* space only, sending it the address of a RAM message will have it fetch values from the same numeric address in the *code* space—certainly the wrong data.

The next example loads a based variable from an element of an array of structures. If you can convince yourself that the resulting assembly code is correct, you are on your way.

---

### Pointer to *xdata* Residing in *data*

```

1  #define uchar unsigned char
2  uchar xdata *data y;
3  typedef struct{uchar x,y;}coord;

```

---

<sup>23</sup>You cannot define the pointer separately. Rather, you define it by naming the based variable. This line says there is a byte variable found where *x* is pointing.

<sup>24</sup>This line and the following three show the compiler-specific keywords of the Keil/Franklin compiler. The first three are specifying where the variable pointed to is located. The pointer will not be a universal one as discussed in the next section. The last definition line sets up a pointer, *zz*, specifically located in off-chip RAM pointing to a byte variable located in on-chip RAM. Where the *pointer* is stored is otherwise established by the choice of the memory model in the compiler invocation or the *#pragma* at the top of the file.

<sup>25</sup>When you pass an array to a function, you usually pass the *address* of the array. You do this where it is called by using the *&* operator which returns the *address* of the variable that follows.

<sup>26</sup>This line actually puts the constant in the place *pointed to* by *x*. Here the constant goes into *count* because *x* (the pointer) was just loaded with the *address* of *count*.

```

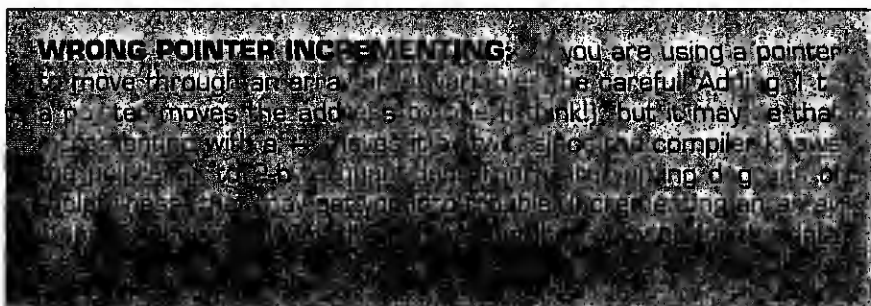
4   coord move[20];
5   void main(void){
6       y=0x6000;
7       *y=move[3].x;
8   }
; FUNCTION main (BEGIN)
; SOURCE LINE # 5
; SOURCE LINE # 6
0000 750060 R      MOV    y,#060H
0003 750000 R      MOV    y+01H,#00H
; SOURCE LINE # 7
0006 850082 R      MOV    DPL,y+01H
0009 850083 R      MOV    DPH,y
000C E500 R      MOV    A,move+06H
000E F0          MOVX   @DPTR,A
; SOURCE LINE # 8
000F 22          RET
; FUNCTION main (END)

```

---

## UNIVERSAL POINTERS

Some compilers have *universal* pointers that include a *third* byte that holds a code identifying which type of memory space is involved.<sup>27</sup> For the address-specific pointers, compilers use either 2-byte pointers or 1- and 2-byte pointers. The 3-byte pointer allows library functions to accept pointers to *any* memory space and then internally decide which set of code to use for processing. Otherwise, the programmer must know which memory space to use and carefully use the right space with the right library function. In all of these situations, you trade code efficiency against flexibility.



<sup>27</sup>Keil/Franklin's universal pointer codes are 1 = *idata*, 2 = *xdata*, 3 = *pdata*, 4 = *data*, and 5 = *code*.

## ARRAY POINTERS

Things rapidly become complicated when you consider pointers to arrays. C can go wild! A *based* array is straightforward enough:

---

```

                                Based Arrays
#define uint unsigned
28 uint xdata a[];
   a[22] = 0xff;

```

---

```

#define uint unsigned
uint xdata ar[];
uint xdata *a;
a=&ar;
29 *(a+22) = 0xff;

```

---

## ARRAYS OF ARRAY POINTERS

Particularly confusing to programmers who started on some other high-level language is the relationship in C between pointers and arrays. The name of the array *is* a pointer. You can have a pointer to an array and you can have an array of pointers! You can even have a pointer to an array of pointers (called *multiple indirection*). In the next example, an array of pointers is used for a group of display message *pieces* that are strung together when sent to the display device. The example keeps all the message pieces as well as the pointer array in *code* space. If some pieces are canned headers while other pieces are current values of variables (in RAM) either use generic (3-byte) pointers or else make an array of structures where one element designates the memory type and a second element is the pointer. With the pointers to the message *pieces* in an array, you can pass the pointer to the pointer array to a display function, which works through the entire string. Although

---

<sup>28</sup>The two examples are *exactly equivalent*, and the preference depends on the notation you want to use in the routine. On large machines it is said that the latter approach compiles to faster code, but that may not be the case with the 8051.

<sup>29</sup>I think that incrementing by 22, *\*(a+22)*, will add 44 to the value in *a* since *a* points to (2-byte) integers. I think the compiler will determine the type of variable involved and increment by the number of bytes per element. Before you stake your career on it though, you ought to check by compiling with the code option turned on (Chapter 9).

this looks quite complicated, if you are pressed to save code space for messages, you can go as far as identifying repeated phrases and reusing the code with this sort of pointer system.

---

### Multiple Indirection

```

#include<reg51.h>
#define uchar unsigned char
uchar code m1[]= {"this is a test"};
uchar code m2[]={ "you failed"};
uchar code m3[]={ "you passed"};
uchar code m4[]={0};
30  uchar code *code fail[]=
    {&m1[0], &m2[0], 0};
uchar code *code pass[]=
    {&m1[0], &m3[0], 0};
void display(uchar code **message){
    uchar code *m;
31    for (; *message!=0; message++){
32        for (m=*message; *m!=0; m++){
            P1=*m;
        }
    }
}
main(){
    display(&pass[0]);
}

```

---

Each array referenced by the array of pointers does not need to be the same size. Some people call this a *sparse array*.<sup>33</sup> For the example here, the compiler trusts that you will keep the indices in bounds by some other

---

<sup>30</sup>The message-piece pointers are collected together into arrays that are also put in code space. This is particularly useful if the RAM is only on-chip, because the pointer array then requires none of that very limited resource.

<sup>31</sup>This line walks through the array of pointers until a zero pointer is found. Since the initial value of message is already set, no initialization is needed in the *for* loop.

<sup>32</sup>This line sets *m* to point to the start of one of the message pieces and then increments *m* through the piece until a null is found indicating the end of that message piece. The code definition with ""s automatically adds a null (0) at the end of the string.

<sup>33</sup>Another form of array, more common in data processing on large computers, is the *linked list*. Here each string of data includes a pointer to the next string, possibly with a number indicating the size of each string. This sort of approach is used with storage on disks.

means.<sup>34</sup> For predefined arrays, C has various library functions such as *sizeof* and *strlen*. Check the compiler manual for more specific details on those functions.

## STRUCTURE POINTERS

It is a very small step from based arrays to based structures. Having left off the discussion with arrays of structures holding arrays, a real application for *based* structures is a message format for multitasking. One task communicates with another task by “sending” a message. You could hand only the message *pointer* to the operating system, which would pass the message (the pointer) to the receiving task. Both tasks must agree on the structure of the message.<sup>35</sup>

---

### Based/pointer Structures

```
#define uint unsigned
#define uchar unsigned char
struct msg1 {uint lnk;uchar
             len,flg,nod,sdt,cmd,stuff;};
struct msg1 *msg;
void rgsendmessage(struct msg1 *m);

main(){
    uchar stuff;
    msg->len=8;
    msg->flg=0;
    msg->nod=0;
    msg->sdt=0x12;
    msg->cmd=0;
    msg->stuff=stuff;
    rgsendmessage(msg);
}
```

---

<sup>34</sup>In message arrays, usually either the length of the message is specified as the first byte of the array, or else a special character is put at the end—usually a nonprinting character that would not otherwise occur in a message. C automatically inserts a *null*, *\0*, at the end of any defined string. It is probably equally efficient to use either an end-marker or an initial-length number, but the former is more flexible if you later revise your messages or string them together.

<sup>35</sup>In this example, you see the structure that DCX and BITBUS systems used. The *operating system* could dynamically allocate the actual storage location out of a memory pool. To go further on dynamic allocation of memory you should buy the companion book!

## UNIONS

Usually included with structures is another data type called a *union*. A union is, as the name implies, a combination of different data types; it applies different names and types for the *same* space. Suppose you want to store a 16-bit timer value that you can only read in as 2 bytes.<sup>36</sup> While you could use a *cast to an integer* and an *8-bit shift* to get the high bits in place, it is also possible to define a *union* made up of a 2-byte structure and an integer. When you want to fill the high byte, you refer to the space as 2 bytes, but when you want to use the result, you refer to the space as an integer.

---

```

                Union of integer and bytes
#define uint unsigned
#define uchar unsigned char
37 union split{uint word;struct{uchar hi;
                uchar low;}bytes};
union split newcount;
38 newcount.bytes.hi=TH1;
newcount.bytes.lo=TL1;
oldcount=newcount.word;

```

---

Going one step further, a union is very useful for message structures an operating system assigns. When a message of the sort described under *Based Structures* arrives, the receiving function must understand the message by a prearranged pattern. If several different message forms are possible (depending on the source of the message), a *union* makes it easier to lay a different *template* over the sequence of bytes depending on its origin. Incidentally, when you say you are using *a based structure made up of a structure and a union of structures*, you will again impress your friends who claim to “know” C! If you mix in arrays of pointers to structures made up of arrays, you may even amaze yourself, but you *can* disentangle it if you understand the basics.

---

<sup>36</sup>The details of timers are covered in Chapter 11.

<sup>37</sup>The first line here sets up the *tag* for a union type called *split*. The parts are *word* and *bytes*—the latter consisting of a structure with 2 bytes called *hi* and *lo*. The only part that becomes implementation specific is the decision that *hi* comes before *lo* which is the case for the normal 8051 arrangement of bytes. This might not transfer to another processor.

<sup>38</sup>The reference to the elements has three parts, all separated by the dot (.). *TH1* and *TL1* are the SFRs that hold the 16-bit timer/counter value as described in Chapter 11.

---

### Union of structures

```

#define uint unsigned
#define uchar unsigned char
39 union mtag{struct{uint keycode;}kmsg; struct{uchar
    cursor; uchar dat[12];}dmsg;}
40 struct msgform {struct header hdr; union mtag
    dat;}*msg;

keyststring[i]=(msg->dat.kmsg.keycode & 1) +'0';
41 msg->dat.kmsg.keycode>>=1;
    msg->hdr.cmd=0x40;
    msg->dat.dmsg.cursor=0;
    for (i=0;i<=10;i++){
42     msg->dat.dmsg.dat[i]=keyststring[i];
    }

```

---

## REVIEW AND BEYOND

1. How many bytes would be set aside for a ten-element *int* array (*uint array[10]*)? Would the low-order bytes be in a group and then the high-order bytes, or would they be byte-pairs? If the array started at location 2020H, where would the 2 bytes of *array[5]* be found?
2. With the 8051, why are arrays of dimension greater than two quite uncommon?

---

<sup>39</sup>This is the definition of the tag for a type of union, here called *mtag*. It consists of two structures named *kmsg* and *dmsg* for the two anticipated data arrangements—which one to use depends on the message source. The two structures are of different sizes. That is immaterial since the whole thing is to be based, but, were it fixed, the *union* would be allocated the larger of the two sizes and the end space reachable only one way.

<sup>40</sup>This line actually defines the message (which is *based*). The use of the tag name simplifies this line, but it would be permissible to substitute the two structures if you preferred. Taking it step-by-step with tags may help the beginning programmer and won't hurt even experienced programmers. *Header* is another structure (not defined here) which holds the message address, length, and other related information common to all messages under the particular operating system.

<sup>41</sup>The use of a *based* structure requires the *->* for the *first* separator, but you can see all the dots involved in reaching into the union (*dat*), into the specific part of the union (*kmsg*), and to the specific part of the structure (*keycode*).

<sup>42</sup>This shows reaching into an array within the structure.

3. Set up a structure to hold coordinate values (say for drawing graphs in x-y space).
4. What are the different memory spaces in the 8051? Can the same address apply to different spaces?
5. What are the possible solutions to the problem of having different memory spaces when pointers are used? What are some of the trade-offs involved?
6. Why would the *printf()* function be more complicated with the 8051 than in normal flat-address-space computers?
7. Explain the difference between *pointers to arrays* and *arrays of pointers*. Give an example of each.
8. Is there any difference between an array and a pointer?
9. Write out examples of a structure and a *based* structure including the way to reference a member of the structure.
10. What are some purposes for unions?





## SECTION II

# Functions, Modules, and Development

---

The three chapters that follow take you from the basics of programming to the modular development used by multiple programmers on large projects. Chapter 7 introduces functions while Chapter 8 shows how separately developed functions can be combined into programs or even put into libraries for general use. Having prepared you with the broad perspective, Chapter 9 describes how the integrated development environment,  $\mu$ Vision, makes it easy to direct the compiling and combining of modules into a single program.



# 7

## Functions

---

Because programmers are always looking for short cuts, finding the same code in several places leads to the thought, *I shouldn't have to write this code over and over*. That is where functions come into play. A function can do such things as produce steps to drive a stepper motor, or convert a number to a displayable (ASCII) form. The details can be “packaged” in one place. You place a *call* instruction in a program when you need a function. When the function finishes, the last instruction you place in the function is a *return*. That causes program flow to resume with the next instruction after the call in the calling program. You can call the same function from somewhere else and use the code of the function over again.<sup>1</sup>

### SUBROUTINES, PROCEDURES, AND FUNCTIONS

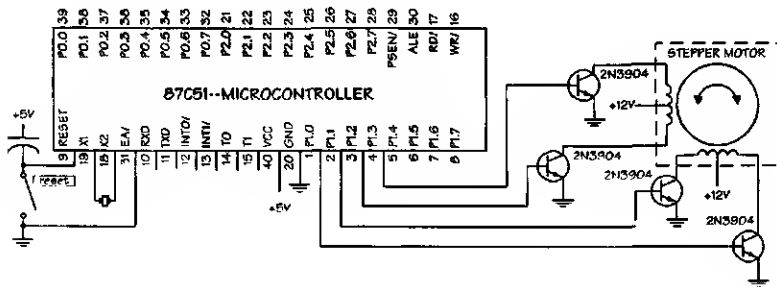
A subroutine, a procedure, and a function can be the same thing. From here on I will use the term *function* since my focus is on C, which prefers that term.<sup>2</sup> Assuming you are a hardware person, I will begin with a function to

---

<sup>1</sup>You can also write functions in different languages. For example, you might write a very time-critical hardware driver in assembly to control the smallest details, but write the main program in C to access math libraries, strings, and easier-to-comprehend code. The details of mixing languages fall under the next chapter

<sup>2</sup>I will use the term *function* generically. Assembly uses the term *subroutine*. C either calls everything a *function* or else differentiates by calling a routine that returns parameters a *function* and a routine that does not return anything a *subroutine* (this is also the convention in BASIC).

drive a stepper motor.<sup>3</sup> First is a hardware schematic and a software flow-chart. Then comes the function—first in C and then in assembly. Each time the *step* function is called it will drive the next phase pattern out to the motor which causes it to move one step (here always in the same direction). The *step* function calls a delay function.<sup>4</sup> After the function comes a separate program piece to show the use of the function.



Stepper driver schematic

### Stepper Driver Function

```
#include <reg51.h>
uchar code pattern[]={0x5,0x9,0xa,0x6};
5 void step(void){
6     static uchar i;
```

<sup>3</sup>The stepper motor used here is a standard *unipolar* four-phase motor. As shown in the schematic, you can drive a unipolar motor with four transistors—one to ground each of the four windings. The alternative, *bipolar* steppers, uses the copper better by having only two windings, but the drivers have to source as well as sink current so an *H Bridge* arrangement is necessary. Such drivers are available in ICs, but they are not as readily available or inexpensive as simple transistors. In addition, it is possible to obtain driver chips that translate to the phase pattern from two lines—one for step and one for direction. Such logic, involving an up/down counter and some gates, is an excellent example for programmable logic devices (PLDs).

<sup>4</sup>Nested functions are discussed later in this chapter.

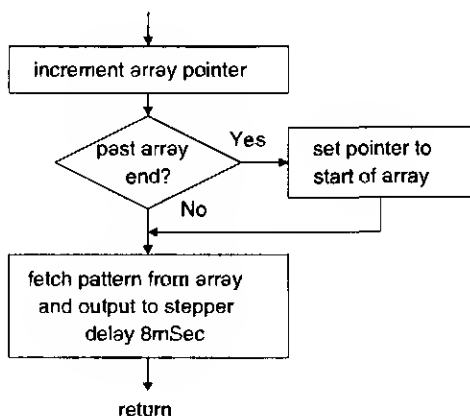
<sup>5</sup>With the advent of ANSI C, the use of the word *void* is encouraged when you do not pass anything *to* or return anything *from* a C function. (Passing and returning are described later in this chapter.)

<sup>6</sup>Make the index *static* to be sure it does not move between steps. The transitions must pick up where they left off if the motor is to move smoothly.

```

7   P1=pattern[i++&3];
8   msec(8);
   }

```



**Stepper driver flowchart**

#### STEPPER DRIVER SUBROUTINE

SPTR SEGMENT DATA

RSEG SPTR

I:DS 1

STEPR SEGMENT CODE

RSEG STEPR

```

9   STEP:MOV A, I

```

```

10  ANL A, #3

```

<sup>7</sup>Here is a typical “condensed” line in C. Because of precedence, the ++ *precedes* the & and the assignment to *i* comes before the actual value is used as the index for the array. The result is that each time this *step* function is used, the value of *i* goes up by one, but if it reaches 4, it is reset to zero. P1 is on-chip port 1. Only the lower four bits are used.

<sup>8</sup>I showed a delay routine in Chapter 5 (page 128). You need a delay between phase transitions to allow the motor to catch up.

<sup>9</sup>An assembly language *subroutine* ought to have a label (here *STEP*) so the *CALL* to it can be simplified. It is *possible* to call to an absolute address, but this would be difficult to manage as lines are added or for relocatable modules. The *RET* pops the return address off the stack to resume the program flow at the line just after the original *CALL*. It is not good to enter a subroutine by anything except a *CALL*. A jump into a routine would not push the return address on the stack so the return would do unexpected things.

<sup>10</sup>The index works through the pattern table, so it must not get above a value of 3. Rather than testing and resetting, since 4 is a nice round binary number, the masking does the resetting in one step and keeps things in bounds even the first time when the value of *I* can be anything.

```

MOV I,A
MOV DPTR,#PATRN
MOVC A,@A+DPTR
MOV P1,A
MOV A,#8
11   LCALL MSEC
RET
12 PATRN:DB 0101B,1001B,1010B,0110B
END

```

---

Here is a part of a program using the *step* function to cause the motor to move twenty steps.

---

```

                        Calling the Function
for(i=1;i<21;i++){
13   step();
}

```

---



---

```

                        CALLING THE SUBROUTINE
MOV R7,#20
14 X:LCALL STEP
    DJNZ R7,X

```

---

<sup>11</sup>I gave this function in Chapter 5 (page 128).

<sup>12</sup>The pattern never changes so put it in the *CODE* segment to burn eventually in ROM with the program instructions.

<sup>13</sup>The call is implicit with the naming of the function. You must show the use of a function by the parentheses although you are not passing any parameters. Sometimes programmers add the word *call* with a *#define call* line at the top to allow the call to appear and improve understandability (ignored by the compiler). For certain 8051 family relatives with small on-chip ROM (750,751), the *LCALL* and *LJMP* are not supported, and the compiler must be given the *ROM(small)* directive to avoid using them.

<sup>14</sup>Assembly has two kinds of calls. The *LCALL* involves 3 bytes for the instruction and allows a full 16-bit location for the subroutine. The *ACALL* is unusual. It fits the call into 2 bytes of instruction but only reaches over the 2k block of the next instruction. That is to say, the call address is made up of the top 5 bits of the current location combined with the bottom 11 bits found in the *ACALL* instruction. While it is efficient for code space in small programs, it is a challenge for the assembler and linker programs when modules may be relocatable. The compiler identifies modules that use this as *inblock* and the linker keeps them at specific 2K-byte boundaries. With most assemblers, you can just use the word *CALL* and the assembler will choose the form of call to use.

## FUNCTIONS EASE UNDERSTANDING

Functions are not just a tool of a lazy programmer. As programs become increasingly complex, *understandability*, more than code savings, is important. Even for a function used only once, you can break it out of the in-line code. Naming it so the call describes the *purpose* of the function gives your reader a quick, general picture of what is going on without becoming lost in the details.

For example, suppose you make a function out of a block of code that gets the mean and standard deviation of a group of readings. Name it *process\_statistics*. When a reader of the program comes to *process\_statistics* in the program, they need not understand all the details to appreciate what is to happen. For more detail, they can go to the function's code. If their interest is in the hardware interface to the A-D converter, obviously *this* is not the function to study.

It has become a rule-of-thumb that no function (or main module) should be longer than about *sixty lines* or about a page.<sup>15</sup> Any code blocks longer than that ought to be broken into other functions so the reader does not have to follow a long run of in-line code. The ultimate disaster is a fifteen-page in-line program with multiple jumps forward or back by several pages. The same rules should apply to flowcharts—fit them each on one page or less and show the details in separate flowcharts.

With a *descriptive name*, the *purpose* of a function is apparent in the call. A function helps keep the focus on the main purpose of the program. When the details are very hardware-oriented, a good name makes it intuitively obvious what sorts of things happen as in *step* and *delay*.

Besides being more understandable to someone reading the program, functions do save code space. Because their code is in only one place and the call to the function takes just 2 or 3 bytes, the space savings with long, repeated functions can be considerable.<sup>16</sup>

---

<sup>15</sup>Some people even argue for twenty lines, which fits on an old CGA CRT display. In the last few years, higher resolution SVGA displays have eliminated that limit so it might better be a forty-line rule!

<sup>16</sup>The process of calling and returning *does* take a little more computer time than *in-line code*, but for a simple call this is only a few microseconds. If the function involves more than one or two lines, the code savings and the increased understandability far outweigh the time cost of the call.



## Drivers

Functions that go between your software and hardware are called *drivers*.<sup>17</sup> They group all the hardware-specific details in one place. For example, to interface to keypad, put all the interface details (for example, scanning, buffering, conversion to binary or ASCII codes) into a driver program module. Other program modules need only request the latest inputs, letting the driver handle the details.

Besides lumping the details in one place and *hiding* them from the rest of the program, drivers make it easy to adjust to hardware *changes*. If you have different sorts of keypads in various projects, develop different drivers for each one and link in the appropriate one for each project.<sup>18</sup> Suppose the final printed board layout requires the switching around of a few bits on one port. If you have carefully used drivers, just changing a few lines of code in the driver makes the software ready to go. You do not need to hunt through the entire code, finding all the places where you used the bits.<sup>19</sup>

## NESTED FUNCTIONS

It is common to have functions make use of other functions. *Nesting* means that one function is called from inside the other function (like a bird in a nest?). To keep within the sixty-line limit and to improve understandability,

---

<sup>17</sup>It is less common to call a *software to software* interface a driver. Just as a hardware driver allows the software using the hardware to interface it in a simple, logical, and standardized way, so software drivers can allow consistent interface between the user's programming and changing software. I can imagine software drivers for specialized processing functions that change depending on the number of points to process or the desired speed or accuracy. Perhaps the software could change depending on whether it used a digital signal processing (DSP) algorithm or a more conventional method. This is getting close to the features that make up C++, but that is beyond the scope of this book. It is still more common to use drivers in relation to hardware, however.

<sup>18</sup>When you pick out prewritten drivers to match the hardware you are *configuring* the system. For large computer systems, drivers come with the system software for many common pieces of hardware—printers, kinds of storage and memory devices, types of displays, etc. This process is well known to PC integrators where the software is configured for the particular disk drives or display hardware installed.

<sup>19</sup>*Embedded* systems (especially the 8051 family of processors) are usually small, control-oriented, with no consistent, generally accepted set of attached hardware. General-purpose drivers seldom exist, and you will probably write your own.

as programs become more complex it is common to nest functions to a depth of four or five.<sup>20</sup>

## PASSING PARAMETERS

As programmers considered the need to avoid repeated code, they realized that much code is *almost* the same. If you do the same thing to one variable this time and to a different variable next time, why not fill in a *separate* variable *in the function*, first with one value and, later, with another value? The function then always works on its own variable holding the value passed to it. With C functions, you can pass several values (*parameters*) and return one value.<sup>21</sup> If you let the compiler do it automatically with the parenthesis of the function call, you are *passing parameters*. This transfer is done in a strictly defined way such as, *the first byte passes in R1, the first integer in R1/R2*, but the compiler handles that.<sup>22</sup>

**CONFUSION ON PASSING ARRAYS** If you want to pass an array to a function, give the name of the array without the brackets as in `display(a)`. If the parameter to be passed is a particular *element* of the array, give the name with a specific number within the brackets as in `show(a[1])`. If you want to pass a *portion* of an array, begin and end with the array name, use the ampersand, and use brackets as in the function call `display(&array[2])`.

<sup>20</sup>A possible limit to nesting with the 8051 family is the size of the stack space in internal memory. Each call puts 2 bytes (the return program-counter address) on the internal stack, so eventually there is no internal stack space. On other processors, C compilers usually rely heavily on the stack for passing parameters, but the 8051 versions of some compilers have options that put all parameters on an artificial *external* stack. Five or ten levels of nesting without parameter passing is usually no problem even for on-chip stack. For small programs, you can safely ignore the nesting and stack depth limits.

<sup>21</sup>You ultimately do this by machine instructions, of course, and you can do it yourself in assembly language. You just move data to the *subroutine's* variables before calling it.

<sup>22</sup>If you are going to combine different languages, or if you want to know the *specific* parameter passing rules for Keil/Franklin C, see the next chapter (page 191).

## EXAMPLE: WRITE TO LCD MODULE

To illustrate passing parameters to a function, I have an example using a liquid crystal display (LCD). The LCD display module I am using is a Seiko M4032 with two rows of forty characters, but it is typical of virtually all the alphanumeric (non-graphical) ones. I show the software with *two* drivers—one for the drive directly from the ports of an 87C751 and the other directly memory-mapped off the expansion bus. The schematics are near the end of Chapter 2 (pages 46 and 47).

### Writing to LCD from Ports

The basic writing sequence from I/O ports for the I/O pins of the LCD module is as follows:

1. Initially set the *ENA* line and the *RS* *low* (for commands) or *high* (for sending display characters). Assume *R/W* is wired high or left high by software (always write out to LCD—never read back, which ignores the use of the busy flag for timing).
2. Write *ENA high* while holding *RS* and *R/W* unchanged.
3. Send the desired data to the eight data lines (0–7).
4. Return *ENA low*, but maintain *RS* and *R/W* unchanged.

This sequence may seem unnecessarily long, but the specification indicates that *RS* and *R/W* must be present for 140 nsec *before* start of *ENA* and the data must *lead the end of ENA* by 320 nsec. Using I/O ports on normal 12 MHz processors, you will never need software delays between the above steps.

### Writing to LCD as Memory

The use of drivers here is unnecessary, but it indicates portable code where you just change the drivers when the hardware changes.<sup>23</sup>

---

<sup>23</sup>With direct interface of an LCD module to an 80C31's expansion bus (shown on page 47), some of the signals are technically too fast for this display device. A number of students using these devices have gone directly to a memory-mapped LCD from a 12 MHz 8051 and find no problems over normal temperature ranges. It would be quite a different matter with the DS520, for example, where the read and write signals are much faster.

### LCD Module Pinout

Pin No.	Symbol	Level	Function
1	Vss	0V	Ground
2	Vcc	+5V	Power Supply
3	Vee	0 to +5V	LCD Drive
4	RS	H/L	Register Select Signal H: Data Input L: Instruction Input
5	R/W	H/L	Read/Write Signal H: Read (LCD $\leftarrow$ micro) L: Write (LCD $\Rightarrow$ micro)
6	E	H $\Rightarrow$ L	Enable Signal (No Pull-up Resistor)
7	DB0	H/L	Data Bus (least significant bit)
8	DB1	H/L	Data Bus
9	DB2	H/L	Data Bus
10	DB3	H/L	Data Bus
11	DB4	H/L	Data Bus
12	DB5	H/L	Data Bus
13	DB6	H/L	Data Bus
14	DB7	H/L	Data Bus (most significant bit)

### LCD Commands

Instruction	Code										Description	Execution Time (max) <sup>1</sup>
	RS	R/W	7	6	5	4	3	2	1	0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Clear display; returns cursor to home position (Address 0)	1.64 msec
Cursor At Home	0	0	0	0	0	0	0	0	1	*	Returns cursor to home position (Address 0); returns display to original position (no change to DDRAM)	1.64 msec
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction; specifies whether to shift display (during data write & read)	40 $\mu$ sec
Display On/Off Control	0	0	0	0	0	0	1	D	C	B	Sets ON/OFF of: all display(D), cursor ON/OFF (C), blink of cursor (B)	40 $\mu$ sec
Cursor/Display Shift <sup>2</sup>	0	0	0	0	0	1	S/C	R/L	*	*	Moves cursor; shifts display without changing DDRAM contents	40 $\mu$ Sec
Function Set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length(DL), # display lines(L), character font(F).	40 $\mu$ sec

Instruction	Code										Description	Execution Time(max) <sup>1</sup>
	RS	R/W	7	6	5	4	3	2	1	0		
CGRAM Address <sup>3</sup>		0	0	0	1	CG Address (6 bits)					Sets CGRAM address, CGRAM data is sent or received after this setting	40 $\mu$ sec
DDRAM Address <sup>4</sup>		0	0	1	DRAM Address (7 bits)					Sets DDRAM address. DDRAM data is sent or received after this setting	40 $\mu$ sec	
Busy Flag/ Address Read		0	1	BF	Address counter					Reads Busy flag (BF), reads address counter contents	0 $\mu$ sec	
CGRAM/ DDRAM Write		1	0	Written Data (8 bits)					Writes data into DDRAM or CGRAM	40 $\mu$ sec		
CGRAM/ DDRAM Read		1	1	Read Data (8 bits)					Reads data from DDRAM or CGRAM	40 $\mu$ sec		
<sup>1</sup> I/D=0: Decrement S=1: With display shift S/C=0: Cursor movement S/C=1: Display shift R/L=1: Shift to the right R/L=0: Shift to the left DL=1: 8-bit N=1: 2 lines N=0: line F=1: 5 $\times$ 10 dots F=0: 5 $\times$ 7 dots BF=0: Instruction acceptable BF=1: Internal operation is being performed CGRAM: Character Generator RAM ACG: CGRAM Address ADD: DRAM Address Corresponds to cursor address AC: Address Counter, used for both DDRAM and CGRAM fcp=250kHz: However, when frequency changes, execution time also changes												

<sup>1</sup>*Time between commands:* Among the instructions are ones to do a status read back from the module (the busy flag as well as the current cursor address) and to read back the data from the display RAM or the character-generator RAM. If you do not read back the status (the case in the example program that follows), allow at least 40  $\mu$ sec after most commands and display writes to be sure the instruction will be accepted. The display *may* be ready sooner but without polling the status (busy flag) you cannot be sure. Unless you poll the busy flag, an arbitrary long delay is the easiest solution. (No damage will occur to the display by writing too quickly, but the results could be erratic.) Even with 100- $\mu$ sec delays between writes, an entire forty-character display is filled in 8 msec, so the writing speed is not slow to an observer. Two of the commands—cursor home and clear display—take longer and should have 1.7 msec to execute as shown in the last column.

<sup>2</sup>A useful mode, not used in the example, is to have the display shift left automatically as you write characters. This gives a scrolling effect that is quite useful for showing long messages on a short display. Position the cursor at the right edge of the display and as each character is written, all the previous characters will move one space to the left until they "fall off" the left edge. Be careful to check the relationship between the multiple rows since they may all move and the one row will probably not flow into the next row. (For 1  $\times$  16 displays the *one* row of sixteen characters is actually *two* blocks of eight separated by unused addresses!)

<sup>3</sup>The character generator RAM allows up to eight characters of your own (codes 0 through 7). Write 8 bytes in succession beginning at CG address 0, 8, 16 . . . for your own patterns of 5  $\times$  7 dots. Address 0, for example, holds the top 5-bit row of character 0, address 1 holds the next row down, etc. Once you have filled up 8 bytes of CG RAM, display that character by sending its code (0–7) to the display, just like you display an "A" by sending a 41h. This is useful if you want non-ASCII symbols such as the up and down arrow.

<sup>4</sup>The cursor can be moved directly. For the 2  $\times$  20 LCD, row two begins at address 64<sub>10</sub> (40<sub>16</sub>). If you want select erasing, remember that a blank is 20<sub>16</sub>—not 0. With the addressable cursor you can keep the display blank and only update individual numbers.

---

**Drivers for Memory-mapped LCD**

```
/*connections of direct-driven LCD */
uchar xdata *LCDcmd;24
uchar xdata *LCDdat;
/*put next two lines in initialize()*/
LCDcmd=0x2000;
LCDdat=0x2001;

void lcddatwr(uchar dbyte){
    *LCDdat=dbyte;
}

void lcdcmdwr(uchar dbyte){
    *LCDcmd=dbyte;
}
```

---

---

**Drivers for LCD Connected to 87C751 Ports**

```
/*put this in initialize()*/
sbit ena=0x81;
sbit rs=0x82;
sbit rd_wr=0x80;
rd_wr=1; /*all writes to LCD*/

void lcddatwr(uchar dbyte){
    rs=1;
    ena=1;
    P1=dbyte;
    ena=0;
}

void lcdcmdwr(uchar dbyte){
    rs=0;
    ena=1;
    P1=dbyte;
    ena=0;
}
```

---

<sup>24</sup>Keil/Franklin software has the absolute access macro XBYTE so the use of the pointers could be replaced by XBYTE[0x2000]=dbyte in the command function and by XBYTE[0x2001]=dbyte in the data function.

## LCD Driver Use Program

```

#include <reg51.h>
#define forever for(;;)
#define uchar unsigned char
#define uint unsigned
void lcddatwr(uchar b);
void lcdcmdwr(uchar b);

25 void display(uchar startloc, uchar *s){
    lcdcmdwr(0x80 | startloc); msec(1);
    while(*s)lcddatwr(*s++); msec(1);
}

26 void numdsp(uchar startloc, uint number, bit dpt){
    lcdcmdwr(0x80 | startloc); msec(1);
    uint m = 10000;
    bit ldgzero = 0;
    do{
        if((((number/m)>0)|| (m == 1))||(m == 100)
            &&(dpt == 1)) ldgzero = 1;
        if (ldgzero == 1) lcddatwr(number/m+'0');
        else lcddatwr(' '); msec(1);
        number %= m;
        if ((dpt == 1) && (m == 100)) {
            lcddatwr('.'); msec(1);
        }
27 }while((m /= 10)>0);

    }

    void msec(uint x){
        uchar j;
        while (x--> 0){
            for (j=0; j<125; j++);
        }
    }

```

<sup>25</sup>This function looks simple despite its ability to handle messages from *code* or *xdata* because, by default, a generic (3-byte) pointer is used and the third byte notifies the function which memory space is to be used.

<sup>26</sup>This function useful, but it is too complex to absorb quickly. It takes an unsigned integer and converts it to a decimal display on the LCD. It suppresses leading zeroes but maintains alignment by substituting blank spaces.

<sup>27</sup>I do not know if you prefer to put the *while* on the next line—presumably the *do* will alert you to look for the *while*.

```

void initialize(void){
    msec(15); /*set up LCD modes*/
    lcdcmdwr(0x30); msec(4);
    lcdcmdwr(0x30); msec(1);
    lcdcmdwr(0x30); msec(1);
    lcdcmdwr(0x38); msec(1); /*1/16 duty?*/
    lcdcmdwr(0x0e); msec(1); /* dsp on*/
    lcdcmdwr(0x01); msec(2);
    lcdcmdwr(0x06)I; msec(1);
}

void main(void){
    uint x=0;
    initialize();

28     display(0,"test");

29     display(64,"time");
    forever{

30         numdsp(69,x++,0);
        msec(1000);
    }
}

```

---

## RETURNING VALUES

Besides sending differing parameters *to* a function, you can get a single value (the “answer” or “result”) returned *from* the function.<sup>31</sup> The normal termination of a function is at the last `}`. If a function is to return a value, the next-to-last line should be *return (expression)*. Any place in a function that you encounter a *return*, the function ends and passes back the parameter if

---

<sup>28</sup>This line sends top row display information.

<sup>29</sup>This moves to the second row of the  $2 \times 40$  display.

<sup>30</sup>This sends the value of *x* for display. Since the count goes up every 1000 msec, it is a seconds count to the accuracy of the delay loop. Other methods are much better for obtaining accurate times, but the goal here is to show the display.

<sup>31</sup>If you have *several* values to return, there is no *direct* way to do it, but you can operate on global variables (see the section later in this chapter on the scope of variables) or return a *pointer* to an array holding the results.

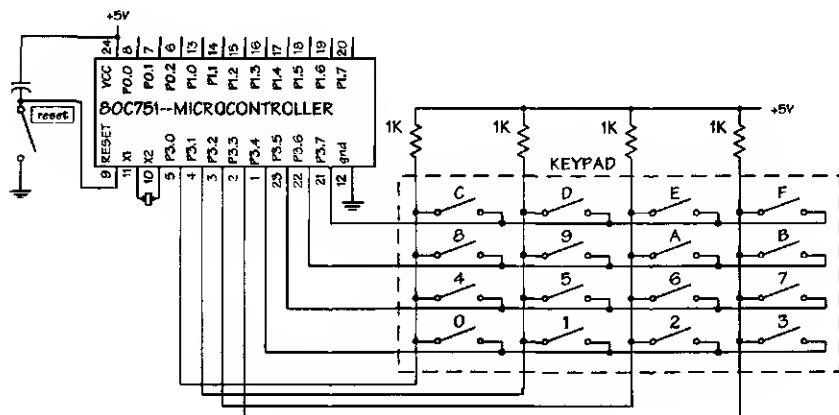


there is one. Program flow leaves the function despite any lines that may follow.

In ANSI C, the type of variable you return comes before the function name. *Void* indicates you return nothing.<sup>32</sup> The appearance of *return x* or *return* will terminate the function, but with a *void* type function, it is normal to omit the *return* and allow the flow to reach the final `}`.

In C, you call a function that returns nothing by writing its name (without “call”) as I just showed. If the function does return a value, the function name comes in an assignment line and the returned value from the function replaces the function name in evaluating the rest of the expression. This is an *implicit call*.

## EXAMPLE: SCAN A KEYBOARD

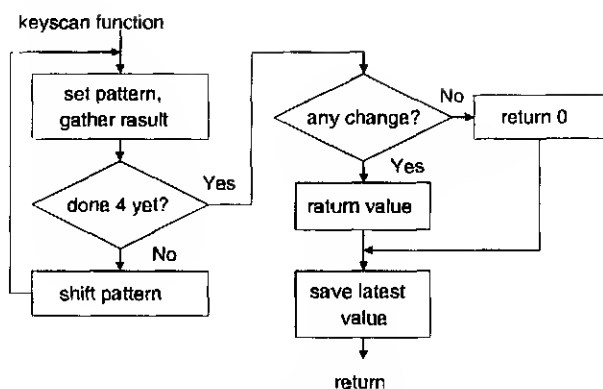


**Keypad schematic**

This example is derived from an earlier one. The function scans the keyboard by driving one column at a time low (putting *pattern* out the top of four bits of *P1*) and reading in the rows from the bottom of the same port. In one loop it scans all four columns one after the other. The function then assembles the result into a 16-bit quantity, checks for changes, and returns.<sup>33</sup>

<sup>32</sup>*Void* was added to the newer (ANSI Standard) C so the compiler could check whether you are using the function correctly. Before that, C compilers assumed that you would return an integer and did not check what sorts of parameters you passed.

<sup>33</sup>It *could* have carried out the pushes/releases test described earlier in Chapter 4, page 102.



**Keyscan flowchart**

### Keyscan Function

```

#include <reg51.h>
#define uint unsigned int
#define uchar unsigned char
34uint oldscan;
uint keyscan(void){
    uchar pattern;
    uint temp;
    uint new_scan=0;
35    for(pattern=0x10;pattern>0;pattern<=1){
        P1=~pattern;
        new_scan = (new_scan<<4) | (P1&0x0f);
    }
36    temp=(new_scan!=old_scan)?newscan:0;
    old_scan=new_scan;
    return temp;
}
  
```

<sup>34</sup>The choice of variable declarations is intentional to allow the variables used only within the procedure to be overlaid. *Old\_scan* can be defined outside the function or defined *static*.

<sup>35</sup>Remember *pattern<=1* means *pattern=pattern<<1*.

<sup>36</sup>Since *old\_scan* must be updated *before* leaving, and since *return* terminates the routine, it is necessary to save the result temporarily before updating and leaving the function. The *?:* operator works like the *if/then/else* operations in that the expression in the (unnecessary) parentheses is evaluated and if true results in the first expression (here *new\_scan*) or if false results in the *0*. In this case, the resulting expression is assigned to *temp*. Remember that the *=* works from *right to left* so the assignment to *temp* comes last. Clearly, if you want more directly understood programs you would avoid this operator or at least add additional parentheses to make the right-to-left more consistent with normal algebra.

**KEYSCAN SUBROUTINE**

```

PERMANENTDATA SEGMENT DATA
RSEG PERMANENTDATA
    OLD_SCAN:DS 2
KEYM SEGMENT CODE
RSEG KEYM
    KEYSCAN: CLR A
            MOV R6,A ;NEW_SCAN IN R6,R7
            MOV R7,A
            MOV R2,#08H ;START COLUMN 1
KS1:MOV A,R2
    RL A ;MOVE TO NEXT COLUMN
    MOV R2,A
    CMP A
    MOV P1,A ;DRIVE COLUMN
    CALL ROT4 ;READY THE RESULT
    MOV A,P1 ;GET RESULT
    ANL A,#0FH
    ORL A,R7
    MOV R7,A
    CJNE R2,#0H,KS1 ; SCAN DONE?
    MOV R2,#0 ;USE AS ='S COUNTER
    MOV A,OLD_SCAN
    CJNE A,6,KS2 ;DANGER-ASSUMES BANK 0
    INC R2 ;ONE 'NO CHANGE'
KS2:MOV OLD_SCAN,R6
    MOV A,OLD_SCAN+1
    CJNE A,7,KS3 ;ASSUMES BANK 0
    INC R2 ;ONE 'NO CHANGE'
KS3:MOV OLD_SCAN,R7
    CJNE R2,#2,KS4;BOTH BYTES SAME?
    CLR A
37    MOV R7,A
    MOV R6,A ;RETURN 0
KS4:RET
38 ROT4: MOV R3,#4 ;ROTATE R6,R7 LEFT BY 4
    RO1:CLR C
    MOV A,R6

```

<sup>37</sup>Consistent with some versions of C, I return the 2-byte result in R6 and R7.

<sup>38</sup>The rotate left by 4 is put into a subroutine since it is used repeatedly and is a clearly defined operation.

```

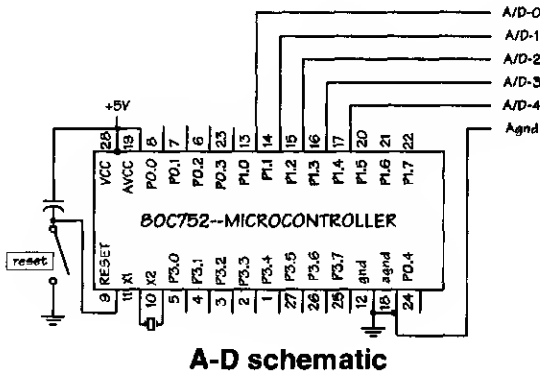
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A
JNC RO2
INC R6;CARRY AROUND
RO2:DJNZ R3,RO1 ;4-BIT ROTATE
RET
END

```

---

## EXAMPLE: READ AN A-D CONVERTER

This next example shows you how you can have the same software for different hardware by changing drivers.



A program using either driver comes first. I show it just collecting ten readings and storing them in an array. It does not matter to the main program which particular driver and hardware are involved.

---

### Using A-D Driver Function

```

#include<reg51.h>
#define uchar unsigned char
#define int unsigned
uint adreading(void);
uint array[10];

```

```

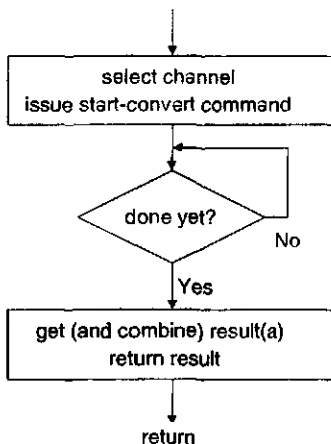
uchar i;
void main(void){
    for(;;){
        for(i=1;i<10;i++){
            array[i]=adreading();
        }
    }
}

```

---

The following driver functions are for *two* analog to digital (A-D) converters. First is a driver for the internal A-D in the 87C752—one of the 8051 family including a built-in, multiplexed 8-input 10-bit A-D converter. The other version, for hardware found on page 44, uses an expanded 80C31 with a separate 10-bit A-D converter. The drivers start the conversion, wait until the result is available, read the result (collect it byte-by-byte and assemble it into an integer), and then return the result to the calling function.<sup>39</sup>

The details of using the two converters are different because of the particular addressing and the nature of the converters. I gloss over aspects of the power and grounding for the converters to reduce noise—the software is the focus. Notice that the drivers hide all the specific details, so changing between devices would only require changes in the driver. You want to work out the details only *once* for a driver like this one!



**A-D driver flowchart**

<sup>39</sup>A multitasking approach would wait for the result in a way that does not tie up the processor—off either a scheduler or hardware interrupts.

**A-D Driver**


---

```

40 #include<reg752.h>
    #define uint unsigned int
    uint adreading (void){
        ADCON=0x20; /*select channel 0*/
        ADCS=1; /*start conversion*/
        while (ADCI==0); /*till done*/
        return ADAT;
    }

```

---

```

41 #include <absacc.h>
    #define uint unsigned int
    #define AD2LO XBYTE[0x4000]
    #define AD2HI XBYTE[0x6000]
    uint adreading (void){
42         sbit ad2busy=P3^2;
        AD2LO=0; /*start-convert*/
        while (ad2busy);
43         return (uint)AD2HI<<8 | AD2LO;
    }

```

---

**A-D DRIVER SUBROUTINES**


---

```

44 AD1 SEGMENT CODE ;FOR 87C752
    RSEG AD1
    ADDR G1:MOV 0A0H,#20H ;SELECT CHANNEL 0
        SETB 0A3H ;START CONVERSION
    L1:JB 0A4H,L1 ;LOOP TILL ADCI HIGH
        MOV R7,084H ;FETCH READING

```

---

<sup>40</sup>This version interfaces the A-D in the 87C752 shown here. The designations for the A-D internal registers are included in the header file. It takes a little time to figure out all the specifics for the additional hardware in other 8051 family members, but the data books are quite thorough.

<sup>41</sup>This works for the A-D of Chapter 2, page 44.

<sup>42</sup>This is a specific method in Keil/Franklin C to set the bit at a specific location—here the top bit of *PI*.

<sup>43</sup>The *cast* to an unsigned integer (*uint*) ensures that the shifts will not assume an 8-bit value. Then the *!* will be with a 16-bit value so the result will be a 16-bit *unsigned int* to be returned. This example had considerably more parentheses for safety (which is not a bad idea anyway), but a close check of the C precedence rules (Chapter 4, page 115) shows that the *cast*, *uint*, precedes the shift *<<* which precedes the *or* *|*.

<sup>44</sup>This works for the on-chip A-D of the 87C752.

```

45      MOV R6, #0
      RET
      END

```

---

```

46      AD2 SEGMENT CODE ;FOR SEPARATE HARDWARE
      RSEG AD2
      ADDR2:MOV DPTR, #4000H
      MOVX @DPTR, A
      L2:JB P3.2, L2 ;END OF CONVERT?
      MOVX A, @DPTR
      MOV R7, A
      MOV DPTR, #6000H
      MOVX A, @DPTR
      MOV R6, A ;SAVE HIGHEST NIBBLE
      RET
      END

```

---

## IN-LINE CODE ALTERNATIVES

In a very few situations, the time it takes to call a function is unacceptable. You can avoid a call by cutting and pasting the instructions each place you need them or you can use a shorthand notation to represent the instructions while keeping the benefits of in-line code.<sup>47</sup> In assembly, use *EQU*s (equate) and in C *#define*. In either case, the assembler/compiler inserts the equivalent text at the time it processes the program, but the *listing* that you see will still have the shorthand name.

## SCOPE OF VARIABLES AND FUNCTIONS

The compiler needs to “know about” all the variables in a program. Several people may write pieces of the overall program and keep their pieces in separate files. Suppose two functions accidentally use the same variable *name*,

---

<sup>45</sup>I discuss parameter-passing conventions in Chapter 8. The assumption here is that a 2-byte value is returned in *R6* and *R7*.

<sup>46</sup>This works for the A-D of Chapter 2, page 44.

<sup>47</sup>One additional possibility in assembly is the use of a *macro*. A macro is somewhat like an equate (*EQU*), but with the possibility of substituting parameters. The result is in-line code where you have “blanks” to fill in differently each time you use the macro, somewhat like passing parameters to a function. Not having used macros after finding they differed significantly among assemblers, I omit further coverage.

say *i* for an index, or *x* for an unknown, intending to refer to things stored at different addresses. There can be trouble! You may have a *scope* of variables problem. The *scope* of a variable is the range of program instructions over which its name has meaning. Outside its scope, that name is either undefined or refers to a *different* variable.

There are definite rules established to avoid scope problems:

1. A piece of program outside the scope of a given variable cannot refer to that variable by *name*.<sup>48</sup>
2. C recognizes variables found at the top of a file anywhere within *that* file.<sup>49</sup>
3. The compiler knows a variable defined *outside* a function only from the point of definition to the end of the file. You are not *forced* to define a variable at the top of the module.
4. You can put the main function *first* and define the variables later, but then you must *prototype* the variable (declare its type and number) before you refer to it.
5. You can use *automatic variables* (defined within one function) only in the function where you defined them. They are subject to overlaying unless you make them *static*.
6. Define any parameters passed into a function in the function; they are unknown outside.

Within an assembly language program file, labels (defined with a colon in the instructions or with an *EQU*) work *anywhere* in that one file; a first assembler pass picks up all the names. You cannot use the same label twice in one file.

---

<sup>48</sup>Such a function might accidentally or intentionally do so by using *based* (pointer) addressing, but that moves your code out of the safety of the compiler's checking. Program revisions could put the variable at a different address if you forgot to change the pointer.

<sup>49</sup>Other files can also know about such variables—see Chapter 8, page 118.



## Scope of Variables

	C	Assembly
<b>Variables in this function</b>		
variable that may be overlaid	Y( ){ int X;}	
variable that must remain	Y( ){ static int X;}	
<b>Variables at outer level of this module</b>		
known in this module only	static int X;	X:DS 2
known to other modules	int X;	PUBLIC X X:DS 2
allocated in other module	extern int X;	EXTRN DATA(X) MOV DPTR, #X MOVX A, @DPTR

**DUAL DEFINING A VARIABLE:** It is a common error to *define* a variable as a passed parameter as well as a global (outside the function) variable. If the variable is global and you want to use it inside a function, you do not need a special definition—just use it by name. The common error is to name a passed parameter the same as the global variable, make changes to the inside parameter, and expect to have changed the global variable. Then you wonder why the outside variable (which you *think* is the same as the inside one) has not changed.

**FORGETTING OVERLAYING:** Any variable declared *within* a function (an automatic variable) is subject to overlaying unless overridden with compiler directives or made *static*. You cannot expect local variables to hold the same value when entering the procedure as when last leaving the procedure. For example, a function counting interrupts must not declare the count variable within the procedure (or else declare it *static*, making it available to anything else).

## REVIEW AND BEYOND

1. What is the “sixty-line rule”? What happens if you violate it?
2. What is a “driver,” and what are its advantages?
3. What can you do when a function needs to return more than one parameter?
4. When is in-line code preferable to functions?
5. What is an implicit call?
6. Write a program using *adreading* along with a separate function to compute a sliding reading of the last ten readings.
7. Referring back to the end of Chapter 6, set up and initialize two arrays of two-element structures holding x-y coordinates for drawing, say, a square. Then write a plotting function with a *based* equivalent to use inside the function. Show how you would call the function to plot a square.

# 8

## Modular Programming

---

### WHY MODULAR PROGRAMMING?

When starting out, most programming students write single-module programs that hold *all* the instructions for the project. To re-use a piece of code, they simply make a copy of an old program and delete any unwanted parts. This saves typing and reduces the chances of making mistakes, but *there is a better way!*

When your programs get longer or when several people are writing pieces of the overall program, *modular programming* is better. Develop your larger project by separately developing software pieces and then *link* the modules together to make up the final program. This approach has several benefits:

1. Modular programming makes program development *more efficient*. Small programs are easier to understand and test than large programs. When you know the expected inputs to the module and the desired outputs, it is more straightforward to test a small module.
2. Modular programming greatly *simplifies debugging* by allowing you to isolate a problem to a specific module.
3. With modular programming, you can keep functions in a *library* for use on the day when the same sort of software requirement reappears in another project. If you need a display driver, take it from the library—do not start all over.

## TERMS AND NAMES

Let me define several terms commonly used in this area before proceeding:

- A *segment* is a connected unit of *code* or data (*data*, *idata*, *xdata*) memory. Segments may be either *relocatable*—their final locations in memory are left up to the link/locate utility—or *absolute*—the programmer specifically assigns addresses.<sup>1</sup>
- A *complete segment* is the combination of the *partial* segments from various modules. For example, the complete *code* segment for a multi-module program is the combination of the code segments for the individual functions and the segment for the main program.
- A *module* is a *file* holding a collection of one or more segments and is given a name by the programmer. Often a module is a collection of related functions for display, calculation, or user interface. A single module can include segments of code *and* various types of data segments. Up to now your programs have each been completely contained in one module.
- A *program* is the goal of the entire development process. In this book, it is a single, absolute module merging all the absolute and relocatable segments from all the associated modules.<sup>2</sup> It is ready to run by downloading or burning in EPROM.
- *Intel HEX format* represents the resulting output code files (from Intel and Franklin, at least) so all the bytes in the file are printable ASCII characters. Another, more compact form, *binary* format, represents each byte of code with a single byte in the file. The Intel HEX format is described in a footnote in Chapter 9.
- A *library* is a file holding one or more modules. Usually the modules are relocatable object modules from the compiler or assembler that you

---

<sup>1</sup>For examples needing absolute modules, consider memory-mapped ports or other hardware—you should never let the linker choose these addresses. Likewise, if you want a field-definable configuration table—perhaps in a separate EPROM—put it at the address of the chip boundary.

<sup>2</sup>This is unlike programs that run under operating systems such as on a PC where most programs are relocatable. Although the hardware of the x86 family can relocate even absolute modules because of the segmentation, most of those programs are compiled so that there are no absolute jumps or calls. The 8051 architecture is not so friendly to relocation since, while there are short relative jumps, the long jumps and calls are absolute and must have absolute addresses before the program can be run. The linker/locator can fill these in for anywhere with relocatable modules, but they will never be *run-time* relocatable.

can combine with other modules at link time. The linker selects out of a library *only* the modules requested (referenced) by other modules. Using one module in the library does not bring in all the code of the library. For example, if, of all the trig functions, you use only *sine*, you may not bring in the code that generates tangents.

- **Header file:** You may have wondered why all my C programs have `#include <reg51.h>` at the start. Technically, it is not necessary! A header is only a shortcut to save you typing lines yourself. Here are the contents of several common header files. Header files are only text—no programs or object files are involved in a header. A close look shows that the header for a C program is entirely *sfr* or *sbit* definitions for the specific byte and bit addresses of the special function registers (internal hardware configuration locations) of the 8051. The *include* file for the assembler sets up definitions for the same things.

---

#### reg51.h (for C compiler)

---

```

/*BYTE Register*/
sfr P0 = 0x80;
sfr P1 = 0x90;
sfr P2 = 0xA0;
sfr P3 = 0xB0;
sfr PSW = 0xD0;
sfr ACC = 0xE0;
sfr B = 0xF0;
sfr SP = 0x81;
sfr DPL = 0x82;
sfr DPH = 0x83;
sfr PCON = 0x87;
sfr TCON = 0x88;
sfr TMOD = 0x89;
sfr TL0 = 0x8A;
sfr TL1 = 0x8B;
sfr TH0 = 0x8C;
sfr TH1 = 0x8D;
sfr IE = 0xA8;
sfr IP = 0xB8;
sfr SCON = 0x98;
sfr SBUF = 0x99;

/*BIT Register*/
/* PSW */
sbit CY = 0xD7;

sbit AC = 0xD6;
sbit F0 = 0xD5;
sbit RS1 = 0xD4;
sbit RS0 = 0xD3;
sbit OV = 0xD2;
sbit P = 0xD0;

/* TCON */
sbit TF1 = 0x8F;
sbit TR1 = 0x8E;
sbit TF0 = 0x8D;
sbit TR0 = 0x8C;
sbit IE1 = 0x8B;
sbit IT1 = 0x8A;
sbit IE0 = 0x89;
sbit IT0 = 0x88;

/* IE */
sbit EA = 0xAF;
sbit ES = 0xAC;
sbit ET1 = 0xAB;
sbit EX1 = 0xAA;
sbit ET0 = 0xA9;
sbit EX0 = 0xA8;

/* IP */
sbit PS = 0xBC;
sbit PT1 = 0xBB;
sbit PX1 = 0xBA;
sbit PT0 = 0xB9;
sbit PX0 = 0xB8;

/* P3 */
sbit RD = 0xB7;
sbit WR = 0xB6;
sbit T1 = 0xB5;
sbit T0 = 0xB4;
sbit INT1 = 0xB3;
sbit INT0 = 0xB2;
sbit TXD = 0xB1;
sbit RXD = 0xB0;

/* SCON */
sbit SM0 = 0x9F;
sbit SM1 = 0x9E;
sbit SM2 = 0x9D;
sbit REN = 0x9C;
sbit TB8 = 0x9B;
sbit RB8 = 0x9A;
sbit TI = 0x99;
sbit RI = 0x98;

```

---

**reg51.INC (for assembler)**

; BYTE Register			; PSW		; IP			
P0	DATA	80H	CY	BIT	0D7H	PS	BIT	0BCH
P1	DATA	90H	AC	BIT	0D6H	PT1	BIT	0BBH
P2	DATA	0A0H	F0	BIT	0D5H	PX1	BIT	0BAH
P3	DATA	0B0H	RS1	BIT	0D4H	PT0	BIT	0B9H
PSW	DATA	0D0H	RS0	BIT	0D3H	PX0	BIT	0B8H
ACC	DATA	0E0H	OV	BIT	0D2H	; P3		
B	DATA	0F0H	P	BIT	0D0H	RD	BIT	0B7H
SP	DATA	81H	; TCON		WR	BIT	0B6H	
DPL	DATA	82H	TF1	BIT	8FH	T1	BIT	0B5H
DPH	DATA	83H	TR1	BIT	8EH	T0	BIT	0B4H
PCON	DATA	87H	TF0	BIT	8DH	INT1	BIT	0B3H
TCON	DATA	88H	TR0	BIT	8CH	INT0	BIT	0B2H
TMOD	DATA	89H	IE1	BIT	8BH	TXD	BIT	0B1H
TL0	DATA	8AH	IT1	BIT	8AH	RXD	BIT	0B0H
TL1	DATA	8BH	IE0	BIT	89H	; SCON		
TH0	DATA	8CH	IT0	BIT	88H	SM0	BIT	9FH
TH1	DATA	8DH	; IE		SM1	BIT	9EH	
IE	DATA	0A8H	EA	BIT	0AFH	SM2	BIT	9DH
IP	DATA	0B8H	ES	BIT	0ACH	REN	BIT	9CH
SCON	DATA	98H	ET1	BIT	0ABH	TB8	BIT	9BH
SBUF	DATA	99H	EX1	BIT	0AAH	RB8	BIT	9AH
			ET0	BIT	0A9H	TI	BIT	99H
; BIT Register			EX0	BIT	0A8H	RI	BIT	98H

There are several customary extensions for program development files. Not all C compilers follow these conventions—there is a enough variation to make it risky to generalize.

---

<b>.PRJ</b>	file holding setup for development environment
<b>.A51</b>	assembly language source files
<b>.C/.C51</b>	C language source files
<b>.LST</b>	shows the assembled code and errors
<b>.OBJ</b>	(relocatable) object modules
	final absolute module (no extension)
<b>.HEX</b>	final module is converted to Intel's printable format
<b>.LIB</b>	libraries
<b>.M51</b>	listings from the linking/locating process
<b>.LNK</b>	the linker output if linking and locating are separate
<b>.h</b>	input files to be included in a source at compile time
<b>.INC</b>	input files to be included in a source at assembly time

---

## SHARING VARIABLES

At the end of Chapter 7, I began a discussion of scope applied to variables. You need to understand the relationship between variables and functions in *separate* modules. When a compiler works on one module, it has no access to other modules, so it must have information about anything it will use or cross-reference from other modules. Shared variables or shared functions exist in only one place and will have the address filled in by the linker/locator utility.

Sharing requires that the assembler/compiler can set up the code to reference bytes as bytes, integers as integers, arrays as arrays, and based as based. The table that follows summarizes the rules.

Scope of Variables		
	C	Assembly
<b>Variables in this function</b>		
variable that may be overlaid	<code>Y() {     int X;}</code>	
variable that must remain	<code>Y() {     static int X;}</code>	
<b>Variables at outer level of this module (file)</b>		
known in <i>this</i> module <i>only</i>	<code>static int X;</code>	<code>X:DS 2</code>
known to other modules	<code>int X;</code>	<code>PUBLIC X X:DS 2</code>
allocated in <i>other</i> module	<code>extern int X;</code>	<code>EXTRN DATA (X) MOV DPTR, #X MOVX A, @DPTR</code>

## C Variable Scope Conventions

In C, a sharp distinction exists between *declaring* and *defining* a variable or function. A *definition* calls for actual storage allocation, whereas a *declaration* (prototype) tells the nature of the variable *without* allocating storage.

Any local C variable defined *within* a function (called an *automatic* variable) could be *overlaid* and changed between function calls. Some compilers keep automatic variables on the stack so they are always lost when the function is exited, while others (for the 8051) observe the nesting sequence of function calls and put automatic variables at the same fixed places for several functions. Automatic variable names have meaning *only* in the specific function.

Defined within a function, a *static* C variable is quite different from an automatic—while the name stays private, the value stays unchanged between function calls.

C variables defined or declared within one file (module) *outside* any function and *before* the functions that use them can be shared among the functions. They are *public* within the entire module but their names are unknown in other modules. Variables defined outside the functions but *after* a function that would use them must have been prototyped before the function.<sup>3</sup>

C variables shared between modules but defined in another module must be prototyped with *extern* to cause a declaration rather than a definition. Essentially, *extern* notifies the compiler to look elsewhere for the variable rather than to allocate storage. So *extern* stops the allocation of space.

In C, variables defined *outside* any function have space allocated at the definition and they are publicly available to *any* function in *any* module. (Of course, if you have not defined them at the top, they will be unknown to other functions even in the given module!) Adding the word *static* to a declaration *outside* any function sets up a shared variable only for functions *within that* module—its name is unrecognized in any other module. This is a way to share a variable within a module having several related functions. It avoids the chance that in some linked, but unrelated, module a programmer may stumble on the same name for a variable. The intermodule, global nature of C can be a problem if used carelessly.

## C Function Scope Conventions

It is a short step to sharing functions. Refer to the next table summarizing scope for functions.

---

<sup>3</sup>The ANSI standard for C suggests that *all* functions be *prototyped* ahead for the *main* function. Then the actual functions come *after* the main function or in another module. This makes for more program lines, but it does fit an idea called *top-down programming*. Putting the actual function definitions *before* the main function always works and avoids some unnecessary program lines. Some C programmers prefer the ANSI suggestion that the main program come first, the other functions come afterwards, and the actual variable declarations come at the end. This means that everything must be prototyped at the top and then also appear near the end where they are “really” created. Personally, I find this cumbersome, but it does fit with the idea that the “most important” stuff comes first. It is a bit like the difference between calculators with an = key and those that use “reverse Polish” notation.



## Scope of Functions/Subroutines

	C	Assembly
known in other modules	Y( ){ ..... };	PUBLIC Y Y:...
known in this module <i>only</i>	static Y( ){ ..... };	Y:...
<i>found</i> in some other module	Y( );	EXTRN CODE (Y) LCALL Y

Functions in C are naturally *global* (public) and can come before or after functions that call them. If a function should be private to one module, it can be defined *static* and will not be callable from any other module.

## Assembly Scope Conventions

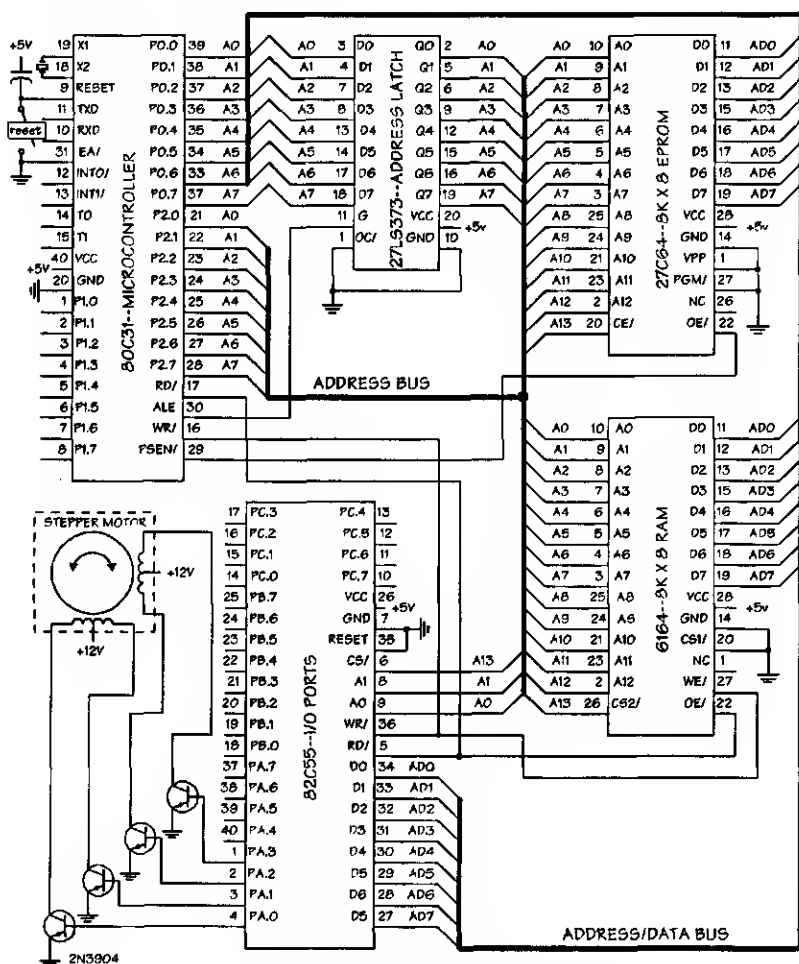
Things are much simpler in assembly. Identify any variable, subroutine, or other label you want to share with other modules as *PUBLIC* in the module where it is defined (at the *top* of the module). In any module that *uses* the public label, list it in an *EXTRN* (no second “e” like *extern* of C) line at the top.

In assembly, subroutines in a module are callable by label anywhere within that module. In its first pass, an assembler gathers up all the symbol names so it can later fill in the values for the *LCALL* or *LJMP*. When specifying *EXTRN*, the type of symbol (*CODE*, *DATA*, *XDATA*, *IDATA*, *BIT*, or *NUMBER*) must be specified so the linker can be sure to keep the same types of things together.

## SINGLE-LANGUAGE MODULES

## Modular ASM51 Example: Stepper Driver

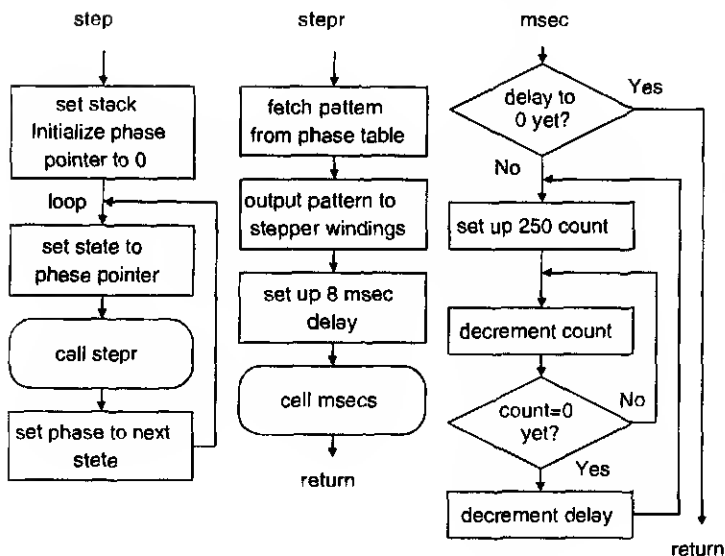
Let’s revisit the stepper motor program from the start of Chapter 7, changing it to a three-module program. First the modules are all written in assembly, then in C, and finally in a mix of both. The hardware, shown in the figure, is attached to an external port.



## Stepper driver schematic

The three *assembly language* program modules, *STEP*, *MSEC*, and *STEPR* illustrate a modular program approach to driving a stepper motor. The  $\mu$ Vision project screen that follows the program listings shows the settings to assemble and then link these modules into an absolute file.

The main program, *STEP.A51*, keeps track of the position of the stepper in *PHASE*, puts it into *STATE*, and calls *STEPR* in some other module.



Step flowchart—ASM51

In all three modules, all the segments are *relocatable* by use of the *SEGMENT* and *RSEG* directive.<sup>4</sup> In general, a serious modular programmer will avoid absolute segments and allow the linker to make efficient use of memory spaces. The three program pieces can be in *separate* files and still work together because the important labels are made *PUBLIC* in one and *EXTRN* in the other. Once you understand naming segments as well as using *PUBLIC* and *EXTRN* in defining the labels, relocatable segments are no problem.

#### STEP.A51 MODULE

LOC	OBJ	LINE	SOURCE
		1	EXTRN CODE (STEPR)
		2	EXTRN DATA (STATE)
		3	MYVAR SEGMENT DATA

<sup>4</sup>Technically the I/O port address is an absolute segment. It is good to publicly define it in one module and make it external in the others because any future hardware address change would then not have to be applied to each module.

```

          4  STEP SEGMENT CODE
          5  STACK SEGMENT IDATA
          6
—         7  RSEG STACK
0000      8    DS 10H
          9
—        10  RSEG MYVAR
0000     11    PHASE: DS 1
        12
—        13  RSEG STEP
0000 750000 14  5START:MOV PHASE,#0
0003 758100 15    MOV SP,#STACK-1
0006 850000 16  LOOP:MOV STATE,PHASE
0009 120000 17    LCALL STEPR
000C 0500   18    INC PHASE
000E E500   19    MOV A,PHASE
0010 5403   20    ANL A,#03H
0012 F500   21    MOV PHASE,A
0014 80F0   22    SJMP LOOP
        23  END

```

## SYMBOL TABLE LISTING

NAME	TYPE	VALUE	ATTRIBUTES
LOOP	CADDR	0006H	R SEG=STEP
MYVAR	DSEG	0001H	REL=UNIT
PHASE	DADDR	0000H	R SEG=MYVAR
SP	DADDR	0081H	A
STACK	ISEG	0010H	REL=UNIT
START	CADDR	0000H	R SEG=STEP
STATE	DADDR	—	EXT
STEP	CSEG	0016H	REL=UNIT
STEPR	CADDR	—	EXT

REGISTER BANK(S) USED: 0

**MSEC.A51 MODULE**

LOC	OBJ	LINE	SOURCE
		1	PUBLIC MSEC
		2	MSECM SEGMENT CODE
		3	

---

<sup>5</sup>By the way, a last-minute change in the hardware involved adding an 8255 port chip, so the code needs to have the chip initialization at the start of *STEP* with *MOV DPTR, #0003H*, *MOVA, #80H*, and *MOVX @DPTR,A*.

```

—          4  RSEG MSEC
0000 6009   5  MSEC:JZ X ;QUIT IF A=0
0002 78FA   6    MOV R0,#250 ;250 X 4=1000
0004 00     7  Z:NOP
0005 00     8    NOP
0006 D8FC   9    DJNZ R0,Z ;4 USEC PER LOOP
0008 D5E0F5 10   DJNZ ACC,MSEC
000B 22    11  X:RET
          12  END

```

## SYMBOL TABLE LISTING

NAME	TYPE	VALUE	ATTRIBUTES
ACC	DADDR	00E0H	A
MSEC	CADDR	0000H	R PUBSEG=MSEC
MSEC	CSEG	000CH	REL=UNIT
X	CADDR	000BH	RSEG=MSEC
Z	CADDR	0004H	RSEG=MSEC

REGISTERBANK(S)USED:0

**STEPR.A51 Module**

LOC	OBJ	LINE	SOURCE
		1	EXTRN CODE (MSEC)
		2	PUBLIC STATE, STEPR
		3	MYROM SEGMENT CODE
		4	MYIRAM SEGMENT DATA
		5	STPPGM SEGMENT CODE
		6	RSEG MYROM
0000	0A	7	TABLE:DB 00001010B,
0001	09		00001001B
0002	05	8	DB 00000101B,
0003	06		00000110B
		9	RSEG MYIRAM
0000		10	STATE:DS 1
		11	
		12	XSEG AT 00000H
0000		13	PORTA: DS 1
		14	
		15	RSEG STPPGM
0000	900000	16	STEPR:MOV DPTR, #TABLE
0003	E500	17	MOV A, STATE
0005	93	18	MOVC A, @A+DPTR
0006	90FFC0	19	MOV DPTR, #PORTA
0009	F0	20	MOVX @DPTR, A; OUTPUT PHASE
000A	7408	21	MOV A, #8



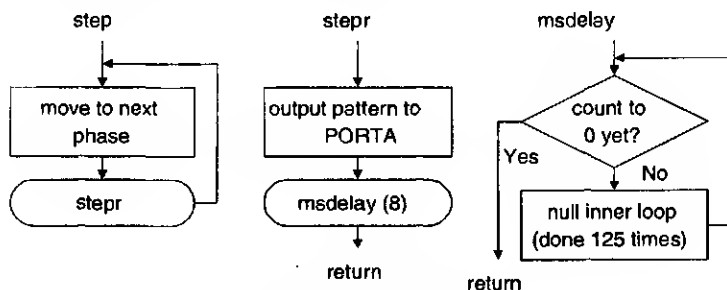
LINK MAP OF MODULE: C:\C51\BIN\MODSTEPA (STEP)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
-----				
* * * * *		D A T A	M E M O R Y	* * * * *
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0001H	UNIT	MYVAR
DATA	0009H	0001H	UNIT	MYIRAM
IDATA	000AH	0010H	UNIT	STACK
* * * * *		X D A T A	M E M O R Y	* * * * *
0000H	FFC0H			*** GAP ***
XDATA	FFC0H	0001H	ABSOLUTE	
* * * * *		C O D E	M E M O R Y	* * * * *
CODE	0000H	0016H	UNIT	STEP
CODE	0016H	0004H	UNIT	MYROM
CODE	001AH	0010H	UNIT	STPPGM
CODE	002AH	000CH	UNIT	MSECM

LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)

**HEX FILE**

```
:1000000075080075810985080912001A0508E508B8
:060010005403F50880F026
:040016000A090506C8
:10001A00900016E5099390FFC0F0740812002A2296
:0C002A00600978FA0000D8FCD5E0F5224F
:00000001FF
```

**Modular C Example—Stepper Driver****Flowcharta for step in C**

The same three example modules, now written in C, come next. If you were sharing variables between modules, you would declare them *extern* in one module to avoid the definition of two *separate* variables.

---

```

                                step.C module
stmt level  source
1           #define uint unsigned int
2           #define uchar unsigned char
3           uchar phase = 0;
4
5           7 stepr (uchar x);
6
7           void main(void){
8           1   8   for(;;){
9           2       stepr(phase=++phase&0x03);
10          2       }
11          1   }
```

```

MODULE INFO:      STATIC  OVERLAYABLE
CODE SIZE =      15  —
CONSTANT SIZE =  —  —
XDATA SIZE =     —  —
PDATA SIZE =     —  —
DATA SIZE =      1  —
IDATA SIZE =     —  —
BIT SIZE =       —  —
```

---



---

```

                                stepr.C module
stmt level  source
1           #include <absacc.h>
2           #define uint unsigned int
3           #define uchar unsigned char
4           #define PORTA XBYTE[0x0000]
```

---

<sup>7</sup>Although the *stepr* function is not in the first module, the compiler knows how to treat it from its *prototype*. Likewise, the *stepr* module can use the *msec* function because it is *prototyped* in *stepr*.

<sup>8</sup>Again, the late addition of an 8255 port chip requires an initialization line here before the loop: *CMD=0x80*; and *#define CMD XBYTE[0x0003]* at the top.



```

5
6      1      void msec (uint x);
7
8      void stepr(uchar state){
9      1      code uchar table []=
10     1      {0x0a,0x09 0x05,0x06};
11     1      PORTA=table[state];
12     1      msec(8);
13     1      }

```

```

MODULE INFO:      STATIC  OVERLAYABLE
CODE SIZE         =  17  —
CONSTANT SIZE     =   4  —
XDATA SIZE        =  —  —
PDATA SIZE        =  —  —
DATA SIZE         =  —  —
IDATA SIZE        =  —  —
BIT SIZE          =  —  —

```

---

#### **msec.C moduls**

```

stmt  level  source
1      #define uint unsigned int
2      #define uchar unsigned char
3      2      void msec(uint x){
4      1      uchar j;
5      1      while (x- > 0){
6      2      for (j=0;j<125;j++){;}
7      2      }
8      1      }

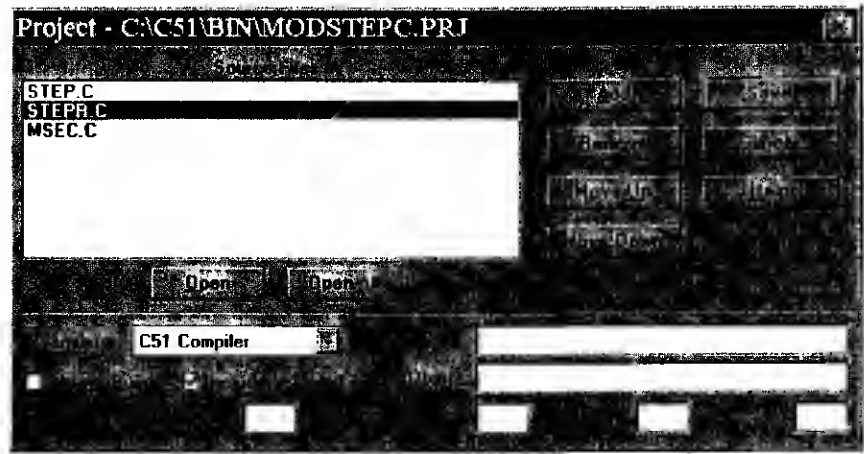
```

```

MODULE INFO:      STATIC  OVERLAYABLE
CODE SIZE         =  27  —
CONSTANT SIZE     =  —  —
XDATA SIZE        =  —  —
PDATA SIZE        =  —  —
DATA SIZE         =  —  —
IDATA SIZE        =  —  —
BIT SIZE          =  —  —

```

---



Project screen<sup>9</sup>

Map file

INPUT MODULES INCLUDED:  
C:\C51\BIN\STEP.OBJ (STEP)  
C:\C51\BIN\STEPR.OBJ (STEPR)  
C:\C51\BIN\MSEC.OBJ (MSEC)  
C:\C51\LIB\C51S.LIB (?C\_STARTUP)  
C:\C51\LIB\C51S.LIB (?C\_INIT)

LINK MAP OF MODULE: C:\C51\BIN\MODSTEP.C (STEP)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
*** **		D A T A	M E M O R Y	*** **
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0001H	UNIT	?DT?STEP
IDATA	0009H	0001H	UNIT	?STACK
*** **		C O D E	M E M O R Y	*** **
CODE	0000H	0003H	ABSOLUTE	
CODE	0003H	000EH	INBLOCK	?PR?MAIN?STEP
CODE	0011H	0004H	UNIT	?C_INITSEG
CODE	0015H	0010H	INBLOCK	?PR?_STEPR?STEPR
CODE	0025H	0004H	UNIT	?CO?STEPR
CODE	0029H	001BH	INBLOCK	?PR?_MSEC?MSEC
CODE	0044H	008CH	UNIT	?C_C51STARTUP
LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)				

<sup>9</sup>Again the three modules (here in C) are listed and the C compiler is the default translator because of the file extension.

---

**HEX file<sup>10</sup>**

```

:03001100010800E3
:0E0003000508E5085403FFF508111580F322E7
:040025000A090506B9
:0F001500EF9000259390FFC0F07F087E00112927
:0100240022B9
:10002900EF1FAA0670011ED39400EA9400400BF466
:0A003900FDEDC3947D50E90D80F742
:01004300229A
:030000000020044B7
:0C004400787FE4F6D8FD75810802008B7F
:10005000020003E493A3F8E493A34003F68001F2C3
:1000600008DFF48029E493A3F85407240CC8C333B1
:10007000C4540F4420C8834004F456800146F6DF80
:10008000E4800B0102040810204080900011E47EFF
:10009000019360BCA3FF543F30E509541FFEE49375
:1000A000A360010ECF54C025E060A840B8E493A33C
:1000B000FAE493A3F8E493A3C8C582C8CAC583CA67
:1000C000F0A3C8C582C8CAC583CADFE9DEE780BF1F
:0100140000EB
:00000001FF

```

---

## MIXING LANGUAGES

If you combine program parts written in *different* languages, you will most likely write the hardware-related functions in assembly. With hardware interfaces, the software details are important. You probably will write the main software in a high-level language where ease of understanding by the reader is most important, and the code is easier to write. A few bytes of extra code used *only once* cost very little in run-time efficiency, but a few bytes used *repeatedly* in a loop can be significant.<sup>11</sup>

---

<sup>10</sup>A quick comparison with the previous assembly version suggests that this is a much larger program (less efficient!), but a closer look shows that the equivalent program is 43<sub>16</sub> bytes to the 36<sub>16</sub> bytes of the previous one. The rest is the startup code to initialize all the on-chip RAM and set the stack.

<sup>11</sup>Compilers have not historically made efficient use of internal registers. A poor compiler might store intermediate results in off-chip RAM, only to pull them back a few lines later because it did not look that far ahead.

Write *everything* in a high-level language and then, when all is working, go back and “fine tune” the critical functions if the potential performance increase justifies the effort. Turn on the *code* option for the frequently-used procedures to see whether you can do better than the compiler.<sup>12</sup> You can break out functions that you feel can be improved and do them in assembly.

## Parameter-Passing Conventions

In mixing languages, your biggest concern is the passing of parameters and returning values. There has to be complete agreement as to how you are doing this if the function is to pick up the passed parameters. Adhere to the conventions in *both* languages. Since *you* have complete control in assembly, make the assembly module fit the high-level language conventions.

So what are these conventions, typically? For Keil/Franklin C, you usually pass all parameters *to* a function at a *fixed* set of locations in internal RAM. If you pass bits, they too must be a sequential string of bits located in internal bit-definable space. Of course, the order and size (char/int) must be consistent between the *calling* and the *called* function. Essentially, you share an identically labeled block of internal RAM between the two. The *caller* fills in the block with the parameters to be passed before issuing the assembly call. The *called* function goes ahead with the assumption that the desired values are already in the block when the call comes.

**Keil/Franklin Default Function Return Conventions**

Return value	Register	Meaning
bit	Carry flag	
byte/char	r7	
word/int	r6, r7	msb in r6, lsb in r7
long	r4-r7	msb in r4, lsb in r7
float	r4-r7	32 bit IEEE format
pointer	r1-r3	msb r2, lsb r1, selector r3

<sup>12</sup>Seeing assembly code is unavoidable with some C compilers that produce *only* assembly language that you must in turn assemble to get the final code. Some compilers also allow inline assembly code to lump those functions in the same file. If you have such a compiler, toggle back and forth between languages all in one file depending on the whim of the moment—that will shake off the casual reader!

C never returns more than a single value. I show the conventions for Keil/Franklin C but may differ for other C compilers. The Keil/Franklin C compiler can pass parameters using registers, using fixed memory locations (like PL/M), or using a stack.<sup>13,14</sup>

## Compatibility by Test

In the end, the easiest way to find out how a given C compiler passes parameters is to compile a dummy function and its call with the *code* list option turned on. You can see exactly what assembly code results and can then model it in your own software. The next six examples illustrate testing for the C parameter passing conventions of the Keil/Franklin compiler.<sup>15</sup> The *differences* in the source files are shown in **bold** type.

---

<sup>13</sup>For most general-purpose C compilers as well as *some* 8051 compilers, the passing is by the stack. That is more consistent with C in general and preserves *reentrancy*. You can enter a *reentrant* function by program flow while it is currently in use. This would only happen if an interrupt came along while the function was in use and the interrupt needed the function as well. Since that could overwrite the registers, a reentrant function must start by saving the current values of all registers on a stack and restoring them at the end. It also must use no fixed-location variables. With the 8051, these stack-intensive functions can be a threat to program integrity since the stack is so small and has no overflow protection. Large functions might even have to generate a synthetic stack in off-chip RAM.

Another applicable word is *recursive*. A recursive function can call itself from within itself. Presumably, any *reentrant* function can be used recursively. Despite the programming elegance of recursion, I do not recommend it with the 8051 because of the havoc it plays with the stack and limited on-chip RAM. Although such an approach is more general, for the 8051 it is much less efficient because, to access external memory (where you could keep a large stack), all activity must be done with one instruction (*MOVX*). This requires setting up and saving the *DPTR* each time. Compilers *can* have reentrant functions use the normal *internal* stack, but that becomes impractical with math library functions, which may consume 100 bytes of stack out of the 128 or 256 available! Other functions may need the on-chip RAM space.

There is *no* portability between C compilers for the 8051 family at the *machine code* level and there are a few different extensions to ANSI C that would have to be changed when changing compilers. Any assembly module written to line up with one compiler probably will not match another.

<sup>14</sup>Obviously, you can write assembly to adhere to *any* desired convention. When mixing two other languages, you might have to write an assembly language *interface* between them. I had to do this when using C with Intel's DCX (BITBUS) hardware that was set up to work with assembly and PL/M.

<sup>15</sup>The default memory model was *RAM(small)*.

---

### Register-based Parameter Passing

```

1  16 #define uint unsigned int
2      #define uchar unsigned char
3      uint fn(uchar chr,uint intg){
4          return 0xc2c1;
5      }
6      uint x;
7      main(){
8          x=fn(0xaa,0xb2b1);
9      }

```

```

17; FUNCTION _fn (BEGIN)
0000 8F00 R MOV chr,R7
0002 8C00 R MOV intg,R4
0004 8D00 R MOV intg+01H,R5
0006 7ECC MOV R6,#0C2H
0008 7FCC MOV R7,#0C1H
000A ?C0001:
000A 22 RET
; FUNCTION _fn (END)

; FUNCTION main (BEGIN)
0000 7FAA MOV R7,#0AAH
0002 7DBB MOV R5,#0B1H
0004 7CBB MOV R4,#0B2H
0006 120000 R LCALL _fn
0009 8E00 R MOV x,R6
000B 8F00 R MOV x+01H,R7
000D 22 RET
; FUNCTION main (END)

```

---

### Alien Parameter Passing

```

1      #define uint unsigned int
2      #define uchar unsigned char
3  18 alien uint fn(uchar chr,uint intg){

```

---

<sup>16</sup>This is the source code with statement numbers added. Actually, I stripped out several things that result from the compilation and linking to save space. One thing omitted is the level of nesting, which is another column in the list file.

<sup>17</sup>This is the result with the code option turned on. It comes from the *compiling*, so the linker has yet to determine the actual addresses of some things.

<sup>18</sup>This is a special case where the C compiler is told to use the *alien* parameter conventions that are the same ones that are used by PL/M. It is available for compatibility and is used in the last example in this chapter. In general, this would not be the preferred method of passing for C.

```

4      return 0xc2c1;
5  }
6  uint x;
7  main(){
8      x=fn(0xaa,0xb2b1);
9  }

; FUNCTION fn (BEGIN)
0000 7ECC    MOV    R6,#0C2H
0002 7FCC    MOV    R7,#0C1H
0004        ?C0001:
0004 22      RET
; FUNCTION fn (END)

; FUNCTION main (BEGIN)
0000 7500AA R    MOV    ?fn?BYTE,#0AAH
0003 7500BB R    MOV    ?fn?BYTE+01H,#0B2H
0006 7500BB R    MOV    ?fn?BYTE+02H,#0B1H
0009 120000 R    LCALL  fn
000C 8E00 R    MOV    x,R6
000E 8F00 R    MOV    x+01H,R7
0010 22      RET
; FUNCTION main (END)

```

---

### Fixed On-chip Parameter Passing

```

1  19 #pragma NOREGPARMS
2      #define uint unsigned int
3      #define uchar unsigned char
4      uint fn(uchar chr,uint intg){
5          return 0xc2c1;
6      }
7      uint x;
8      main(){
9          x=fn(0xaa,0xb2b1);
10     }

; FUNCTION fn (BEGIN)
0000 7ECC    MOV    R6,#0C2H

```

---

<sup>19</sup>This is the same as the first *except* I turned on *NOREGPARMS*. Notice that *main* loads the parameters into on-chip RAM at fixed (to be determined by the link/locate utility) locations rather than into the registers in the previous example.

```

0002 7FCC    MOV  R7,#0C1H
0004        ?C0001:
0004 22      RET
; FUNCTION fn (END)

; FUNCTION main (BEGIN)
0000 7500AA R  MOV  ?fn?BYTE,#0AAH
0003 7500BB R  MOV  ?fn?BYTE+01H,#0B2H
0006 7500BB R  MOV  ?fn?BYTE+02H,#0B1H
0009 120000 R  LCALL fn
000C 8E00    R  MOV  x,R6
000E 8F00    R  MOV  x+01H,R7
0010 22      RET
; FUNCTION main (END)

```

---

### Off-chip Paramatar Passing

```

1  #pragma NOREGPARMS
2  #define uint unsigned int
3  #define uchar unsigned char
4  20 uint fn(uchar chr, uint intg) large{
5      return 0xc2c1;
6  }
7  uint x;
8  main(){
9      x=fn(0xaa,0xb2b1);
10 }

```

```

; FUNCTION fn (BEGIN)
0000 7ECC    MOV  R6,#0C2H
0002 7FCC    MOV  R7,#0C1H
0004        ?C0001:
0004 22      RET
; FUNCTION fn (END)

; FUNCTION main (BEGIN)
0000 900000 R  MOV  DPTR,#?fn?BYTE
0003 74AA    MOV  A,#0AAH
0005 F0      MOVX  @DPTR,A
0006 A3      INC  DPTR

```

---

<sup>20</sup>With the Keil/Franklin C compiler, at least, it is possible to force the parameters of *fn* to off-chip RAM either by using the large memory model [*RAM(large)*] for the specific function as shown here, or using the *LARGE* model overall. Then the parameter loading before the function call would be more complex.



```

0007 74BB      MOV  A,#0B2H
0009 F0       MOVX @DPTR,A
000A A3       INC  DPTR
000B F0       MOVX @DPTR,A
000C 120000 R  LCALL fn
000F 8E00 R   MOV  x,R6
0011 8F00 R   MOV  x+01H,R7
0013 22       RET
; FUNCTION main (END)

```

---

### Reentrant On-chip Parameter Passing

```

1  #define uint unsigned int
2  #define uchar unsigned char
3  21 uint fn(uchar chr,uint intg)reentrant{
4      return 0xc2c1;
5  }
6  uint x;
7  main(){
8      x=fn(0xaa,0xb2b1);
9  }

```

```

; FUNCTION _?fn (BEGIN)
0000 1500 E  DEC  ?C_IBP
0002 1500 E  DEC  ?C_IBP
0004 A800 E  MOV  R0,?C_IBP
0006 A604     MOV  @R0,AR4
0008 08      INC  R0
0009 A605     MOV  @R0,AR5
000B 1500 E  DEC  ?C_IBP
000D A800 E  MOV  R0,?C_IBP
000F A607     MOV  @R0,AR7
0011 7ECC     MOV  R6,#0C2H
0013 7FCC     MOV  R7,#0C1H
0015         ?C0001:
0015 0500 E  INC  ?C_IBP
0017 0500 E  INC  ?C_IBP
0019 0500 E  INC  ?C_IBP
001B 22      RET
; FUNCTION _?fn (END)

```

---

<sup>21</sup>The function can be designated *reentrant* to force parameter passing by a stack. With the *SMALL* memory model (in use here by default), the stack will be an on-chip stack. Notice that it is *not* the actual hardware-supported stack using *PUSH* and *POP* instructions.

```

; FUNCTION main (BEGIN)
0000 7CBB    MOV    R4,#0B2H
0002 7DBB    MOV    R5,#0B1H
0004 7FAA    MOV    R7,#0AAH
0006 120000 R    LCALL    _?fn
0009 8E00    R    MOV    x,R6
000B 8F00    R    MOV    x+01H,R7
000D 22      RET
; FUNCTION main (END)

```

---

### Reentrant Off-chip Parameter Passing

```

1  22 #define uint unsigned int
2    #define uchar unsigned char
3    uint fn(uchar chr,uint intg) large reentrant{
4        return 0xc2c1;
5    }
6    uint x;
7    main(){
8        x=fn(0xaa,0xb2b1);
9    }

```

```

; FUNCTION _?fn (BEGIN)
0000 90FFFE    MOV    DPTR,#0FFFEH
0003 120000 E    LCALL    ?C_ADDXBP
0006 EC        MOV    A,R4
0007 F0        MOVX    @DPTR,A
0008 A3        INC    DPTR
0009 ED        MOV    A,R5
000A F0        MOVX    @DPTR,A
000B 90FFFF    MOV    DPTR,#0FFFFH
000E 120000E    LCALL    ?C_ADDXBF
0011 EF        MOV    A,R7
0012 F0        MOVX    @DPTR,A
0013 7ECC        MOV    R6,#0C2H
0015 7FCC        MOV    R7,#0C1H
0017           ?C0001:
0017 900003    MOV    DPTR,#03H
001A 120000 E    LCALL    ?C_ADDXBP

```

---

<sup>22</sup>Here is an example that uses an off-chip artificial stack for parameter passing. Notice that an external subroutine actually manipulates the stack. In addition, it is not clear where this stack will be located. Why is there the loading of *DPTR* with *#0FFFE*? I would determine where the stack ends up (by disassembling the final located code) before trusting the function completely.

```

001D 22      RET
; FUNCTION _?fn (END)

; FUNCTION main (BEGIN)
0000 7CBB     MOV  R4,#0B2H
0002 7DBB     MOV  R5,#0B1H
0004 7FAA     MOV  R7,#0AAH
0006 120000 R  LCALL _?fn
0009 8E00 R   MOV  x,R6
000B 8F00 R   MOV  x+01H,R7
000D 22      RET
; FUNCTION main (END)

```

---

Where the passing conventions of the earlier examples are compatible, the linking is as simple as picking the object module names. Realize that even with compatible parameter passing conventions, not all companies' software tools produce compatible object file formats; you may be limited by the same company's compiler/assembler when mixing with their other languages. In general, obtain all the tools from the same source or have good guarantees of compatibility before you spend your money.

### Mixed Language Example: Stepper Driver

Having developed multimodule programs in assembly and then in C, let's next mix languages. The pages that follow show *step* in C, and *STEPR* and *MSEC* in assembly. The most logical mixed language arrangements have the "high-level" programming in C and the drivers in assembly.

If you look closely, the C *step* module passes a single byte to *stepr*, which according to the conventions and by checking the *code* listings, it does in *R7*. That being the case I have to change *STEPR*—quite easy, by the way—to pick up the phase pointer from *R7* rather than from public variable, *STATE*. You have to make the assembly agree with what the C compiler expects. The only other change is the name for the function—the Keil/Franklin C compiler appends an underline before the name, so I had to switch *STEPR* to *\_STEPR* in the assembly module.<sup>23</sup>

---

<sup>23</sup>I include the list files so you can see all the information you can derive from even very simple programs. You will find it a useful skill to be able to extract the information you need from the overabundance of irrelevant surrounding material.

**step.lst**

DOS C51 COMPILER V5.10, COMPILATION OF MODULE STEP  
 OBJECT MODULE PLACED IN STEP.OBJ  
 COMPILER INVOKED BY: C51.EXE STEP.C CD SB DB OE  
 PL(69) ROM(COMPACT)

```

stmt  level  source
1      #define uint unsigned int
2      #define uchar unsigned char
3      uchar phase = 0;
4
5      24 void stepr (uchar x);
6
7      void main(void){
8          1      25 for(;;){
9              2          stepr(phase=++phase&0x03);
10             2      }
11         1      }

```

## ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 7
; SOURCE LINE # 8
0000      ?C0001:
; SOURCE LINE # 9
0000 0500      R  INC      phase
0002 E500      R  MOV      A,phase
0004 5403      ANL      A,#03H
0006 FF        MOV      26 R7,A
0007 F500      R  MOV      phase,A
0009 120000    E  LCALL    _stepr
; SOURCE LINE # 10
000C 80F2      SJMP      ?C0001
; SOURCE LINE # 11
000E 22        RET
; FUNCTION main (END)

```

<sup>24</sup>Note that while *stepr* is the function name *here*, down below the compiler named it *\_stepr*, so the corresponding module in assembly must be changed accordingly.

<sup>25</sup>Again, the late addition of an 8255 port chip requires an initialization line here before the loop: *CMD=0x80*; and *#define CMD XBYTE[0X0003]* at the top.

<sup>26</sup>The R7 passes the phase, so you must modify the assembly module accordingly.

NAME	CLASS	MSPACE	TYPE	OFFSET	SIZE
=====	=====	=====	=====	=====	=====
_stepr . .	EXTERN	CODE	PROC	-----	----
phase. . .	PUBLIC	DATA	U_CHAR	0000H	1
main . . .	PUBLIC	CODE	PROC	-----	----
MODULE INFORMATION:    STATIC OVERLAYABLE					
CODE SIZE	=	15	----		
CONSTANT SIZE	=	----	----		
XDATA SIZE	=	----	----		
PDATA SIZE	=	----	----		
DATA SIZE	=	1	----		
IDATA SIZE	=	----	----		
BIT SIZE	=	----	----		
C51 COMPILATION COMPLETE.    0 WARNING(S),    0 ERROR(S)					

**STEPR.LST**

```

DOS MACRO ASSEMBLER A51 V5.07c
OBJECT MODULE PLACED IN STEPR.OBJ
ASSEMBLER INVOKED BY: A51.EXE STEPR.A51 XR DB EP
LOC  OBJ          LINE      SOURCE
                                1  EXTRN CODE (MSEC)
                                2  PUBLIC _STEPR
                                3  MYROM SEGMENT CODE
                                4  STPPGM SEGMENT CODE
                                5
-----
                                6  RSEG MYROM
0000 0A09                7  TABLE:DB 00001010B,00001001B
0002 0506                8      DB 00000101B,00000110B
                                9
-----
                                10 XSEG AT 0000H
0000                    11  PORTA: DS 1
                                12
-----
                                13 RSEG STPPGM
0000 900000    F  14  27 _STEPR:MOV DPTR,#TABLE
0003 EF                15  28 MOV A,R7
0004 93                16  MOV A,@A+DPTR
0005 90FFC0            17  MOV DPTR,#PORTA
0008 F0                18  MOVX @DPTR,A;OUTPUT PHASE

```

<sup>27</sup>The underline preceding the name is necessary to fit the naming convention of the step module that uses this function.

<sup>28</sup>This is a change from the previous assembly version because, rather than a shared public variable, the passing uses *R7*.

```

0009 7408          19      MOV A, #8
000B 120000      F  20      LCALL MSEC
000E 22          21      RFT
                22      END
XREF SYMBOL TABLE LISTING
NAME      TYPE      VALUE      ATTRIBUTES      REFERENCES
MSEC . . C ADDR  ----      EXT          1# 20
MYROM. . C SEG   0004H      REL=UNIT      3#  6
PORTA. . X ADDR  0000H A          11# 17
STPPGM . C SEG   000FH      REL=UNIT      4# 13
TABLE. . C ADDR  0000H R SEG=MYROM      7# 14
_STEPR . C ADDR  0000H R SEG=STPPGM     14#
REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)

```

---

**MSEC.LST**

```

DOS MACRO ASSEMBLER A51 V5.07c
OBJECT MODULE PLACED IN MSEC.OBJ
ASSEMBLER INVOKED BY: A51.EXE MSEC.A51 XR DB EP
LOC OBJ      LINE      SOURCE

```

```

29 1  PUBLIC MSEC
    2  MSEC.M SEGMENT CODE
    3
---- 4  RSEG MSEC.M
0000 6009  5  MSEC:JZ X ;QUIT IF A=0
0002 78FA  6      MOV R0, #250 ;250 x 4 = 1000
0004 00    7  Z:NOP
0005 00    8      NOP
0006 D8FC  9      DJNZ R0, Z ;4 uSEC PER LOOP
0008 D5E0F5 10     DJNZ ACC, MSEC
000B 22    11  X:RET
    12  END

```

```

XREF SYMBOL TABLE LISTING
NAME      TYPE      VALUE      ATTRIBUTES      REFERENCES
ACC. . . D ADDR  00E0H      A          10
MSEC . . C ADDR  0000H      R      SEG=MSEC.M  1 5# 10
MSEC.M. . C SEG   000CH      REL=UNIT      2#  4
X. . . . C ADDR  000BH      R      SEG=MSEC.M  5 11#
Z. . . . C ADDR  0004H      R      SEG=MSEC.M  7#  9
REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)

```

---

<sup>29</sup>Since this links with the assembly version of *STEPR*, no changes are needed.

**stepmodm.M51-mixed languagee**

MS-DOS BL51 BANKED LINKER/LOCATER V3.52, INVOKED BY:  
BL51.EXE C:\C51\BIN\STEP.OBJ, C:\C51\BIN\STEPR.OBJ,  
C:\C51\BIN\MSEC.OBJ TO C:\

>> C51\BIN\MODSTEPM IX NOOL RS (64) PL (68) PW (78)

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:

C:\C51\BIN\STEP.OBJ (STEP)

C:\C51\BIN\STEPR.OBJ (STEPR)

C:\C51\BIN\MSEC.OBJ (MSEC)

C:\C51\LIB\C51S.LIB (?C\_STARTUP)

C:\C51\LIB\C51S.LIB (?C\_INIT)

LINK MAP OF MODULE: C:\C51\BIN\MODSTEPM (STEP)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
* * * * * D A T A M E M O R Y * * * * * *				
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0001H	UNIT	?DT?STEP
IDATA	0009H	0001H	UNIT	?STACK
* * * * * X D A T A M E M O R Y * * * * * *				
	0000H	FFC0H	*** GAP ***	
XDATA	FFC0H	0001H	ABSOLUTE	
* * * * * C O D E M E M O R Y * * * * * *				
CODE	0000H	0003H	ABSOLUTE	
CODE	0003H	000FH	INBLOCK	?PR?MAIN?STEP
CODE	0012H	0004H	UNIT	?C_INITSEG
CODE	0016H	0004H	UNIT	MYROM
CODE	001AH	0010H	UNIT	STPPGM
CODE	002AH	000CH	UNIT	MSECM
CODE	0036H	008CH	UNIT	?C_C51STARTUP

SYMBOL TABLE OF MODULE: C:\C51\BIN\MODSTEPM (STEP)

VALUE	TYPE	NAME	C:0003H	LINE#	
-----	MODULE	STEP	C:0003H	LINE#	9
C:0000H	SYMBOL	_ICE_DUMMY__	C:000FH	LINE#	10
D:0008H	PUBLIC	phase	C:0011H	LINE#	11
C:0003H	PUBLIC	main	-----	ENDPROC	MAIN
-----	PROC	MAIN	-----	ENDMOD	STEP
C:0003H	LINE#	7			

-----	MODULE	STEPR	-----	MODULE	MSEC
C:0016H	SEGMENT	MYROM	C:002AH	SEGMENT	MSECM
D:0009H	SEGMENT	MYIRAM	C:002AH	PUBLIC	MSEC
C:001AH	SEGMENT	STPPGM	D:00E0H	SYMBOL	ACC
C:001AH	PUBLIC	_STEPR	C:0035H	SYMBOL	X
X:FFC0H	SYMBOL	PORTA	C:002EH	SYMBOL	Z
C:0016H	SYMBOL	TABLE	C:002AH	LINE#	5
C:001AH	LINE#	16	C:002CH	LINE#	6
C:001DH	LINE#	17	C:002EH	LINE#	7
C:001FH	LINE#	18	C:002FH	LINE#	8
C:0020H	LINE#	19	C:0030H	LINE#	9
C:0023H	LINE#	20	C:0032H	LINE#	10
C:0024H	LINE#	21	C:0035H	LINE#	11
C:0026H	LINE#	22	-----	ENDMOD	MSEC
C:0029H	LINE#	23			
-----	ENDMOD	STEPR			

#### INTER-MODULE CROSS-REFERENCE LISTING

NAME . . . . .	USAGE	MODULE NAMES
?C_INITSEGSTART.	CODE;	** L51 GENERATED **
?C_START . . . . .	CODE;	?C_INIT ?C_STARTUP
?C_STARTUP . . . . .	CODE;	?C_STARTUP STEP
MAIN . . . . .	CODE;	STEP ?C_INIT
MSEC . . . . .	CODE;	MSEC STEPR
PHASE. . . . .	DATA;	STEP
_STEPR . . . . .	CODE;	STEPR STEP
LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)		

### Mixed Language Example: Math

Here is a second example of an assembly module interfaced to another language, a 32-bit math library I wrote. It is a set of specific functions (originally designed to link with PL/M) for a pre-defined calculation where 16 bits was not sufficient.<sup>30</sup> The first part has two versions due to a differing parameter-passing convention in the C examples. The difference relates to naming and to passing in of parameters. A usage example in C follows the assembly modules.

<sup>30</sup>Actually, with C you could just choose *unsigned long* variables and the linker will bring in the library. I actually wrote these functions, before C was available, to satisfy a specific need for a double-precision algorithm to extrapolate and scale a set of data points. It would be interesting to see if the approach is more or less efficient than the general purpose algorithm that comes with the C compiler.



---

**Assembly math module**

```

31 PUBLIC DADD,DSUB,DMUL,DDIV,DLOAD,_DLOAD
PUBLIC DRESULT,?DLOAD?BYTE
DSTACKA SEGMENT DATA
DSTACKB SEGMENT DATA
DSTACKC SEGMENT DATA
DIMATH SEGMENT CODE

32 RSEG DSTACKA
    STKA:DS 4

RSEG DSTACKB
    STKB:DS 4

RSEG DSTACKC
    STKC:DS 4

RSEG DIMATH

33 _DLOAD:MOV R0,#STKB
        MOV R1,#STKC
        ACALL QMOV ;B TO C
        MOV R0,#STKA
        MOV R1,#STKB
        ACALL QMOV ;A TO B
        MOV R0,#STKA
        CLR A ;ZERO UPPER 2
        MOV @R0,A
        INC R0
        MOV @R0,A
        INC R0

```

---

<sup>31</sup>The key interface functions are *DLOAD* (which picks up the 2 bytes of *A/N* and pushes them onto an arbitrary internal stack of three 4-byte values) and *result* (which pops the arbitrary stack and leaves off the lower two bytes in *R6* and *R7*). Incidentally, there would be an additional difference for other C compilers that do not return using *R6*, *R7*.

<sup>32</sup>The arbitrary 4-byte stacks for this math function could be put into one segment quite nicely, but an early application was using scattered fragments of main memory and these smaller segments could be moved around by the linker to "pack" better.

<sup>33</sup>The Keil/Franklin C compiler insists on having the underline as the first character of any function that passes parameters by register. Only with *#pragma NOREGPARAMS* or *extern alien* will it go to the fixed-location passing and drop the underline from the name. Incidentally, for *reentrant* functions the naming convention is *\_?function* for Keil/Franklin. To add to the fun, some other compilers put the underline in front of *everything*. Needless to say, trials are recommended.

```

        MOV @R0,6 ;COMES IN R6,R7
        INC R0
        MOV @R0,7
        RET

; PUTS ONE 4-BYTE STACK VALUE
; OVER ANOTHER ONE—MOVES IT
QMOV: MOV R2,#4
QMO1: MOV A,@R0 ;FROM @R0
      MOV @R1,A ;TO @R1
      INC R0
      INC R1
      DJNZ R2,QMO1
      RET

DRESULT:MOV R7,STKA+3 ;POPS STACK
        MOV R6,STKA+2 ;RESULT IN R6,R7
        MOV R0,#STKB
        MOV R1,#STKA
        ACALL QMOV ;B OVER A
        MOV R0,#STKC
        MOV R1,#STKB
        ACALL QMOV ;C OVER B
        RET

DADD: MOV R0,#STKA+3 ;LSB OF A
      MOV R1,#STKB+3 ;LSB OF B
      MOV R2,#4 ;4 BYTES TO PROCESS
      CLR C ;NO CARRY INTO FIRSTADD
DAD1: MOV A,@R0
      ADDC A,@R1 ;A+B
      MOV @R0,A ;SAVE IN A
      DEC R0 ;MOVE TO NEXT HIGHER BYTE
      DEC R1
      DJNZ R2,DAD1 ;4 TIMES
      MOV R0,#STKC
      MOV R1,#STKB
      ACALL QMOV ;MOVE C OVER B
      RET

DSUB: MOV R0,#STKA+3 ;LSB OF A
      MOV R1,#STKB+3 ;LSB OF B
      MOV R2,#4 ;4 BYTES TO PROCESS
      CLR C ;NO BORROW FROM FIRST SUBTRACT

```

```

DSU1: MOV A,@R0
      SUBB A,@R1 ;A-B
      MOV @R0,A ;SAVE IN A
      DEC R0 ;MOVE TO NEXT HIGHER BYTE
      DEC R1
      DJNZ R2,DSU1 ;4 TIMES
      MOV R0,#STKC
      MOV R1,#STKB
      ACALL QMOV ;MOVE C OVER B
      RET

```

```

DMUL: MOV A,STKA+3
      MOV B,STKB+3
      MUL AB ;AL X BL
      MOV R2,A ;TEMP BYTE 0
      MOV R3,B ;TEMP BYTE 1
      MOV A,STKA+2
      MOV B,STKB+2
      MUL AB ;AH X BH
      MOV R4,A ;TEMP BYTE 2
      MOV R5,B ;TEMP BYTE 3
      MOV A,STKA+2
      MOV B,STKB+3
      MUL AB ;AH X BL
      ACALL ADDIN
      MOV A,STKA+3
      MOV B,STKB+2
      MUL AB ;AL X BH
      ACALL ADDIN
      MOV STKA+3,R2 ;TEMP RESULTS IN A
      MOV STKA+2,R3
      MOV STKA+1,R4
      MOV STKA,R5
      MOV R0,#STKC
      MOV R1,#STKB
      ACALL QMOV ;PUT C OVER B
      RET

```

```

ADDIN: ADD A,R3 ;LOW PART FROM MUL AB
      MOV R3,A
      MOV A,B
      ADDC A,R4 ;HIGH PART FROM MUL
      MOV R4,A
      CLR A

```

```

        ADDC A,R5    ;IF ANY CARRY UP
        MOV R5,A
        RET

;IF NO BORROW THEN PUTS 1 IN BOTTOM OF A
;THEN SHIFTS A LEFT BY 1
;ANSWER FILLS IN BOTTOM OF A AS IT SHIFTS
DDIV: MOV R2,#17    ;A(32 BIT)/B(LOWER 16 BITS)
DDI1: CLR C
        MOV A,STKA+1
        SUBB A,STKB+3
        MOV R3,A
        MOV A,STKA
        SUBB A,STKB+2
        JC DDI2      ;TOO SMALL-DON'T USE
        MOV STKA,A
        MOV STKA+1,R3
DDI2: CPL C
        MOV R0,#STKA+3
        MOV R3,#4
DDI3: MOV A,@R0      ;SHIFT A LEFT ONE BIT
        RLC A
        MOV @R0,A
        DEC R0
        DJNZ R3,DDI3
        DJNZ R2,DDI1
        MOV R0,#STKC
        MOV R1,#STKB
        ACALL QMOV
        RET
END

```

---

### C usags of assembly math modules

```

#define uint unsigned int
34 void DLOAD(uint sixteen);
        /*push word onto stka-top fills with 0s*/
void DADD(void); /* a+b -> a (32 bit)*/
void DSUB(void); /* a-b -> a (32 bit)*/
void DMUL(void); /* axb -> a (16x16=32bit)*/

```

---

<sup>34</sup>These function prototypes are necessary so the compiler knows how to prepare to use them. From the tests with the *code* option, the use of *alien* will pass by location and return a word in R6,R7. Without the assembly object modules on hand, the *linker* will complain that it was unable to locate *DLOAD*, *DADD*, etc.

```

void DDIV(void); /* a/b -> a (32/16=16bit)*/
uint DRESULT(void);
    /*return lower 16 bits of a*/
void main(void){
    uint a,b,c,d;
    a=999;
    b=1000;

35  DLOAD(a);
    DLOAD(a);
    DMUL(); /*999*999*/
    DLOAD(b);
    DLOAD(b);
    DMUL(); /*1000*1000*/
    DSUB();
    c=DRESULT(); /*1999*/

36  dload(b);
    dload(a);
    dload(a);
    dmul();
    ddiv();
    d=dresult(); /*998*/
}

```

---

## LIBRARIES

Libraries are collections of program modules (functions) that a linker can connect to other programs by the linker. Libraries can hold pre-defined code as with the various C libraries that support the more elaborate C functions. They can also be collections of your favorite *custom* functions written in assembly or C that you use so regularly that you don't want to type them over and over. By putting object modules in the library you can save compile time—they are there ready to be included.

---

<sup>35</sup>This example only tests the use of large numbers to be sure the various carries and borrows are working. With the simulator working in hex notation, it was necessary to determine that 998=03e6, 999=03e7, 1000=03e8, and 999\*999=0f3a71. A modern scientific calculator usually has these conversion functions built in.

<sup>36</sup>Because the assembly math has a stack (reverse Polish) architecture (like many of the math chips) and has no swapping on the stack, it is necessary to get *b* onto the stack *before* doing the *a \* a* operation.

## The Standard C Libraries

It is particularly common with C to use libraries supplied with the compiler. String and floating-point math functions can be quite lengthy. Good embedded software tools include only what is necessary for the specific application. If the linker finds unsatisfied references in the main program module, it searches through modules and libraries listed for the code to include—say the floating-point cosine algorithm. The library and their modules do not need to be in any particular order. A reference can be made in one library module to a (public) symbol found in another module.

If you decide you need the more complex functions such as string or trig functions, most C compilers support them. Depending on the compiler, this might either be automatically included or might require including a header file at the top of the relevant modules that prototypes the additional functions. When the linker puts your program together with the missing functions, given that the needed libraries are included in the project file and available, the pieces will fit properly. For example, if you include a header file named *trig.h*, you could have a line like *float sine(float x);* and in the linker setup you should include something like *c51fps.lib*. Merely including the header file in your source code would satisfy the *compiler* (it would know the meaning of the functions or variables found in the library) but would give link-time errors.<sup>37</sup>

**FLOATING-POINT VARIABLES:** Be careful about using float variables with most of the 8051 family because, with no co-processor, the linker will include large code blocks from the floating-point libraries and it may take a long time to download during development as well as take up an excessive amount of code space at runtime.

## Your Own Library

Re-typing often-used functions is a waste of your time. The previous pages showed you how to include various modules in a project, but the compiler had to convert each module to an object module before linking. The following pages walk you through steps to develop *your own* individualized *library* of (object) functions that you can use in just the same way as you use standard C library functions. Just as a normal library has many hooks

---

<sup>37</sup>If for some reason you wanted to write your own *sine* function rather than using the one in the floating-point library, put your function/subroutine in a module or library of your own as described in the next section.

(or the standard C library has many functions) and you read only the ones of interest, so your software libraries can hold more functions than any particular program might need. The linker will “check out” only the modules it needs.

### Functions for a Library

We have kept the *stepr* and *msec* modules separate until the end. That is easier for debugging because you can get at the individual files to make corrections until all the *publics* are right and the parameter passing is working properly. Writing a library module then only requires referencing the working module in the new project. For less-modular functions you may have to add the appropriate parameter passing and move variable definitions (both automatic and global) inside the module.

Make your functions as general-purpose as possible. Name them so the invocations make good sense when you see them in calling programs. Any shared variables must have their names made *public* in the module and any assembly modules to be used as C functions must have the correct naming and parameter-passing conventions.

### Keep Variables Private

Be careful to either keep all the variable declarations *inside* the function (*automatic* variables) or else label them as *static* variables (*static unsigned char x;*) *outside* the function so the names will not be shared between modules. By defining a *public* global variable (*unsigned char x;*) in a library function module but *outside* the function, errors arise if someone defines the *same* public global name (*x*) in a program.

If you need to observe any of the new library function's variables during initial troubleshooting, put their declarations *outside* the function, since automatic variables may exist only on the stack, making them messy to examine during debugging

### Function Prototypes in a Header File

Function prototypes are necessary in C so the compiler knows how to treat a function that is somewhere else than in the present file.<sup>38</sup> To avoid re-typing the prototypes for the functions coming from your library, put them in

---

<sup>38</sup>The equivalent in assembly is the inclusion of *EXTRN* in programs so the symbols are available but unsatisfied until the linker makes the connection. You must be sure to use the symbols correctly in the program—*MOV DPTR, SYMBOL* if the symbol is the location of off-chip *storage* but *LCALL SYMBOL* if it is the location of a separate *function*.

your own header file. For this example the *msec* and *stepr* function prototype is put into *myfns.h* along with the defines from before (only the necessary ones are shown here). Additional *defines* or prototypes do not hurt anything. As long as you do not *call* a prototype function, the linker will not include it.

---

#### Header File *myfns.h*

```
#include <c:\c51\inc\reg51.h>
#define uchar unsigned char
#define uint unsigned int
void msec(uint x);
void stepr(void);
```

---

Now, with a custom header and library, the *entire* stepper drive program can look like this:

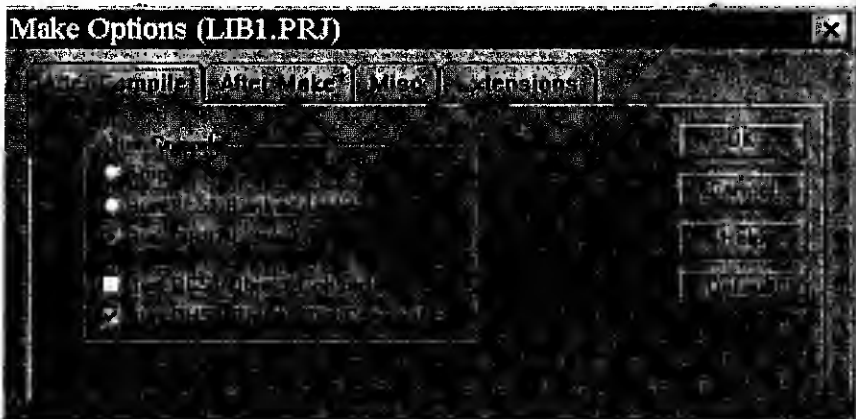
---

#### Final *step.C* Program with Library

```
#include <myfns.h>
uchar phase = 0;
void main(void){
    for(;;){
        stepr(phase=++phase&0x03);
    }
}
```

---

### Making the Library



Calling for the librarian



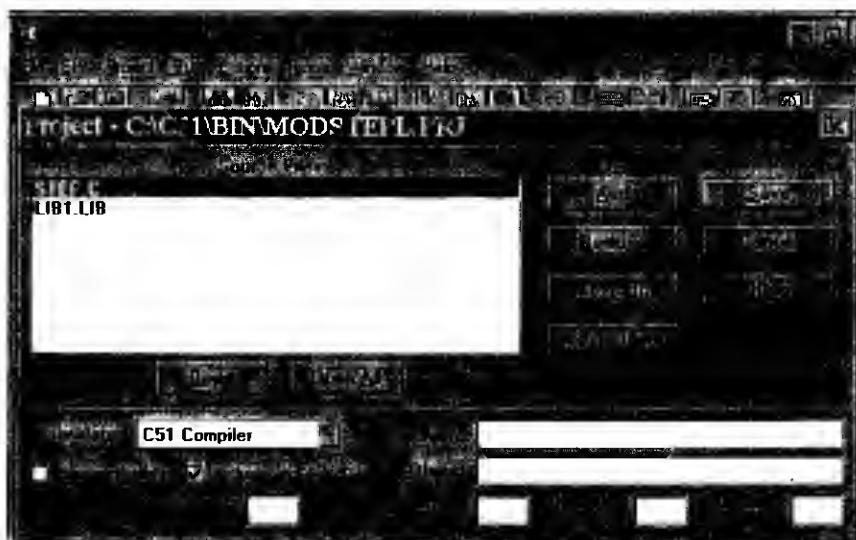
With uVision,<sup>39</sup> you can gather your object modules together in a library so easily it is embarrassing! All you have to do is make a new project, list the modules to go in the library, and check the box that calls for the librarian rather than the box for the linker. Name the *project* with the name you want for the *library* and build the project. In the pages that follow, I put the two assembly modules of the *STEP* program into a library, named *LIB1.LIB*. Then I build a new *STEP* project using only the *step.c* module and the new library. The results are identical to the previous multilanguage project, but the project build time is much less because the latter two modules are in the library in pre-assembled (object) form.



The lib1 project elements

This produces *lib1.lib*, which is unreadable in a normal text editor because it has object code. The next step is to put together a new project that uses *step.c* and the new library:

<sup>39</sup>Again, these development tools are covered in Chapter 9, so skip over this the first time and come back if you need it later. It is "out of order" because the library and this example are logical extensions of modular programming, which is a logical extension of functions, which is a logical extension of the earlier programming material....



STEP project using lib1 library

**modstepm.M51**

```

MS-DOS BL51 BANKED LINKER/LOCATER V3.52, INVOKED BY:
BL51.EXE C:\C51\BIN\STEP.OBJ, LIB1.LIB TO
C:\C51\BIN\MODSTEPL.NOO LRS (64) PL
>> (68) PW (78)
MEMORY MODEL: SMALL
INPUT MODULES INCLUDED:
C:\C51\BIN\STEP.OBJ (STEP)
LIB1.LIB (STEPR)
LIB1.LIB (MSEC)
C:\C51\LIB\C51S.LIB (?C_STARTUP)
C:\C51\LIB\C51S.LIB (?C_INIT)
LINK MAP OF MODULE: C:\C51\BIN\MODSTEPL (STEP)

```

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
***	***	D A T A	M E M O R Y	***
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0001H	UNIT	?DT?STEP
IDATA	0009H	0001H	UNIT	?STACK
***	***	X D A T A	M E M O R Y	***
0000H	FFC0H		*** GAP ***	
XDATA	FFC0H	0001H	ABSOLUTE	

```

* * * * * C O D E M E M O R Y * * * * *
CODE 0000H 0003H ABSOLUTE
CODE 0003H 000FH INBLOCK ?PR?MAIN?STEP
CODE 0012H 0004H UNIT ?C_INITSEG
CODE 0016H 0004H UNIT MYROM
CODE 001AH 000FH UNIT STPPGM
CODE 0029H 000CH UNIT MSEC
CODE 0035H 008CH UNIT ?C_C51STARTUP
SYML TABLE OF MODULE: C:\C51\BIN\MODSTEPL (STEP)
VALUE TYPE NAME
---- MODULE STEP
C:0000H SYMBOL _ICE_DUMMY_
D:0008H PUBLIC PHASE
C:0003H PUBLIC MAIN
---- PROC MAIN
---- ENDPROC MAIN
C:0003H LINE# 7
C:0003H LINE# 8
C:0003H LINE# 9
C:000FH LINE# 10
C:0011H LINE# 11
---- ENDMOD STEP
INTER-MODULE CROSS-REFERENCE LISTING
NAME . . . . . USAGE MODULE NAMES
?C_INITSEGSTART. CODE ** L51 GENERATED **
?C_START . . . . CODE ?C_INIT ?C_STARTUP
?C_STARTUP . . . CODE ?C_STARTUP STEP
MAIN . . . . . CODE STEP ?C_INIT
MSEC . . . . . CODE MSEC STEPR
PHASE. . . . . DATA STEP
_STEPR . . . . . CODE STEPR STEP
LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)

```

As you can see, there is no problem at all and the link map shows all the functions connected in the same way as when the modules were separate.

## SHORTCUTS

### Code Efficiency

The reason to mix languages is to improve program efficiency. You need to decide what you mean by efficient code. Is it the code that takes up

the least memory, the code that runs in the least instruction cycles, or the code that took the least time and effort to write? Fortunately, those objectives are not mutually exclusive! In general, code that takes fewer bytes will run faster, but the choice of looping versus straight line code can make a difference. It is often said that assembly coding can be more efficient than high-level programming, but that is assuming that size and speed of final code are the measure rather than the time it takes to write and debug!

All the talk of code efficiency assumes you have barely enough processor time to get everything done, or that it matters how quickly the processor finishes some task. Most embedded systems spend the majority of their processor time waiting for something to do. A program that only toggles a bit every second will, if it runs faster or more efficiently, just spend more time waiting for the next toggle time. You should consider your specific application before deciding how much effort to put into efficiency. You ought to write in high-level language for things that run only occasionally and involve user interaction. If you still insist, go to assembly for any small, tight, often-repeated loops. These improvements in efficiency are multiplied by the repetition of the loop. Probably those loops involve either hardware or specific math operations, which can perhaps be done more efficiently than in the libraries.

Now that you have seen some examples of compiling with the code option turned on, you can carry out your own investigations of efficiency. There are such wide variations between resultant code depending on the choice of compiler that it is probably useless to get specific. If you study the code you will see that compilers follow a fairly formal set of rules about retrieving and returning variables between lines. Depending on the compiler, there will be more or less foresight in evidence relative to keeping things in registers or on-chip RAM between program lines. A large amount of the code may be moving things around. If you write in assembly, you can carefully plan the register usage to maximize efficiency because you can look far enough ahead to see that you will need some things again. The compiler may not do that. Probably it is with efficiency that the choice of a particular C compiler has the biggest impact.<sup>40</sup>

Some choices that you as the programmer can make have significant impact on efficiency:

---

<sup>40</sup>One example run a number of years ago with four different compilers (provided by Franklin) varied from 220 bytes plus library calls down to 34 bytes with no calls. This is probably a rapidly changing field and would need to be investigated carefully if efficiency is important.

1. Choose the *small* memory model where space allows. This avoids the use of the *MOVX* instruction.
2. With the *large* model, give careful thought to which variables to keep in *data* space. Put there the often-used ones or ones holding intermediate results.
3. Revise the *sequence* of your program operations to finish with one variable before you work with the next one.
4. When you use a *for(;;)* loop recognize that a *DJNZ* instruction is slightly more efficient than a *CJNE* instruction—make the iterative loop count *down*.<sup>41</sup>
5. Use shifting and rotating rather than multiplication and division. For example, a shift left by one has the same effect as a multiplication by two.
6. Use masking by logical ANDing (&) which is much more efficient than using a mod (%) operator.
7. Carefully choose data storage and array sizes based on the inherent binary nature of the computer—multiples of two might slightly simplify index manipulation.

Obviously, these things may not make a large difference and you may need to experiment with the code option turned on to see what is best with your compiler. The more you learn of assembly language and binary math operations, the more efficient you can become in your C programming choices.

### Headers for Register and I/O Definitions

The 8051 hardware and Keil/Franklin C in particular offers nonstandard *extensions* to ANSI C. For Keil/Franklin C these include the port and register definitions found in header files that you should put at the top of your programs (such as `#include <REG51.H>`, `#include <REG552.H>`, or `#include <REG751.H>`).

It seems pointless to include *STDIO.H* because printers, keyboards, and CRT consoles are not usually part of 8051 systems. I do not recommend *MATH.H* because floating-point variables and trig functions result in very big programs. If you *must* have floating-point variables, you will have to link in a different library and probably take a speed penalty.

---

<sup>41</sup>Some of the compiler optimization levels will automatically switch things around this way, but complex expressions may defeat the optimization—develop a good programming habit rather than depending on a clever compiler.

## Off-chip Variables

You may keep variables (and code) in on-chip or off-chip storage. Be alert for the *xdata* and *code* directives when you define variables. When compiling (or via a *#pragma*) you can choose where the default variables will be kept (*SMALL*, *COMPACT*, or *LARGE*).

## Overlaying

**FORGETTING OVERLAYS ON AUTOMATIC VARIABLES:** A variable declared within a procedure may not be the same the next time the procedure is entered. In some versions of C compiler, depending on optimization levels, the variables declared within a procedure might have changed because of overlays.

With very limited on-chip memory space, the linker/locator will normally reuse locations when they are no longer needed by a function. That is to say, if one function neither calls nor is called (even indirectly) from another function, then one function will never be running before the other has finished. Then it can keep the automatic variables in the exact *same* RAM space, much as you reuse registers. You *can* do this sort of thing in assembly by intentionally using the same named variables or by using absolute addresses, but the linker will manage this only for C modules.<sup>42</sup> Overlaying could result in considerably less RAM space requirement if there are several unrelated functions.

## More About Linking

**Linking**, discussed more in Chapter 9, is the process of tying together all the segments of the modules to produce a complete program. The linking software scans the modules to identify all the public symbols (variable, procedure/function, and label names). Once those go into a list, unsatisfied references to them begin having relative addresses filled in—the value for each *LJMP X*, *LCALL X*, or *MOV DPTR,#X* can start to be filled in. *Start to*, is the term because the linker phase does not assign absolute addresses; rather

---

<sup>42</sup>The assembler has no way to determine the nesting level of subroutines since you might write a very non-structured assembly language program. If you want to re-use a variable, *you* have to keep track of whether it could already be in use.

it assigns addresses *relative to* other similar types of *code* or *data*. In other words, the linker first fills in all the references relative to the start of an overall segment by combining all the segments of the same type. It combines all the smaller code segments such as the *main* and individual functions into one big code segment. The small segments can come from several modules as well as from libraries. For example, all the external RAM variables segments are laid out so no segment overlaps another segment. In the locator phase, the software figures out the absolute addresses.

### Segment types

INBLOCK	Segments which use <i>ACALL</i> —must fit in a 2048-byte <i>block</i> <sup>1</sup>
INPAGE	Segments that must fit on one <i>page</i> (the same upper 8-bit address) due to the nature of the internal jumps and calls
BITADDRESSABLE	Segments that must be put in the internal RAM space that is specifically bit addressable
UNIT	Segment that can begin at any byte or bit boundary

<sup>1</sup>None of the linkers are clever enough to allow part of an *INBLOCK* module to cross a block boundary even though calls or jumps *within* module might not cross a boundary!

The way in which the linker fits segments together follows a definite sequence. It combines all the partial segments of the same type (*code*, *xdata*, etc.) into a single segment.<sup>43</sup> It cannot put some segments just anywhere due to the nature of the instructions, as shown in the table above.

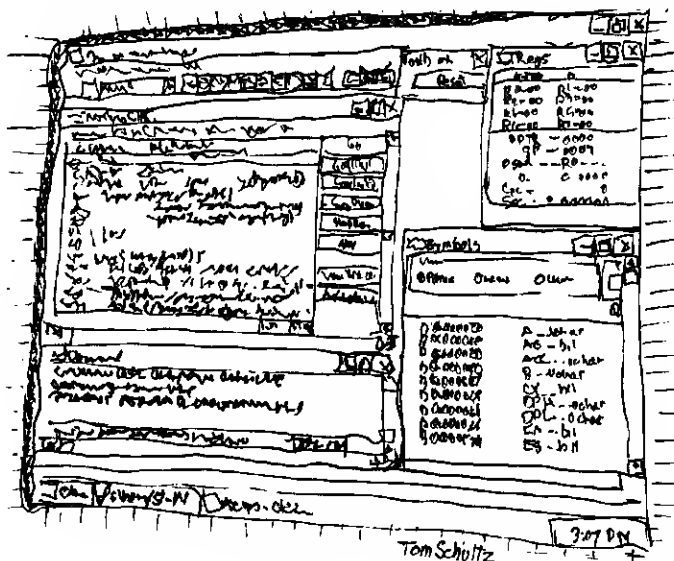
## REVIEW AND BEYOND

1. What are reasons for using modular programming?
2. In the two languages, what is the scope of a variable defined at the top of a module without any special directives?
3. What must you do in both modules to enable one module to call a function in another module?
4. What form of program normally goes into a library? What happens if the library holds more modules than the linker needs?

<sup>43</sup>Technically, it is possible to produce separate segments of similar types, but the need to call out the segments by name complicates the process. When segments need to be located away from pre-committed addresses (as when you are downloading code to a target that has a monitor in ROM), it is easiest to generate an absolute dummy module that “claims” the territory. By listing the dummy module first in the link invocation line, the module stakes out the necessary space and the linker puts the rest of the segments in the remaining space. If the order were reversed, the linker would assign the code, for example, beginning at 0 and would then flag a conflict when the absolute module was encountered that wanted the same space.

# 9

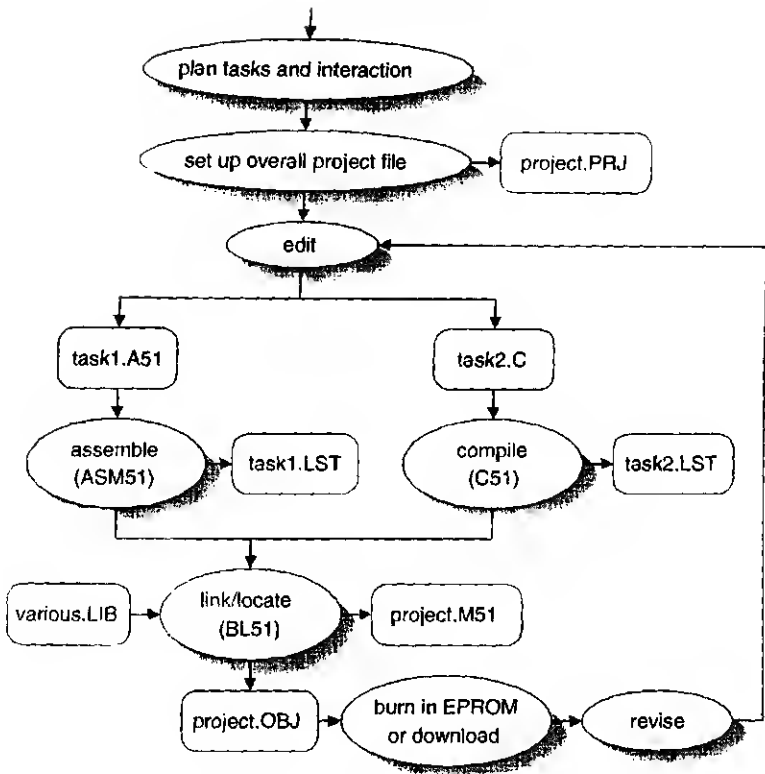
## Development and Debugging



Before going into the details of  $\mu$ Vision, I would like to discuss first the program development steps for embedded applications.<sup>1</sup> The next figure charts these steps for multiple modules, but except for the use of the librarian, the same development process applies for single-module programs.

<sup>1</sup>If you are using the  $\mu$ Vision “integrated development” tools, the steps are merged together in a way that was inconceivable ten years ago. I can remember a time perhaps twenty years ago when a “three-pass assembler” meant you had to feed the source text into the computer *three times* via paper tape. Even a few years ago, you would have had to develop a DOS batch file if you wanted to follow the compiling process with the linking process with one command. Now you can simply ask the environment to “do its thing” and it will re-compile any changed source files and link the entire program in one step.





**The development process**

## THE OVERALL DEVELOPMENT SEQUENCE

You will usually repeat this process many times, because few things work perfectly at first. Plan to get pieces working before integrating them into a finished package. The process is as follows:

1. *Plan* the overall project. This includes picking the particular hardware and planning how you will divide the software up to get the job done.
2. *Write* the software, and type it into files for assembling or compiling.<sup>2</sup>

<sup>2</sup>I personally find it better to plan out the software with a flowchart and some rough code pieces on paper—*pseudo-code* or *P code*—before going to the keyboard, but I recognize that some programmers prefer to just start typing. I have no complaint as long as the typing is preceded by or includes the planning—unfortunately many of the *type-first* programmers take off in random directions and discover only later that they have entered large pieces of wasted programming.

3. *Compile* and/or *assemble* the source programs. This may include putting the object modules into libraries.
4. Get the resulting files (usually *relocatable* with all the jump addresses yet to be filled in) put together and set up at specific memory locations. This is *locating* and *linking*. The latter happens if there are several pieces to the overall program, or if there are libraries to include.<sup>3</sup>
5. Get the resulting *absolute* file into the computer which will be doing the control job. There is often a step to convert the very condensed—one-byte-per-byte of instruction—file into the printable hex format that most EPROM programmers recognize. Getting the program into the target microcontroller can either be done by *downloading* for development where there is a resident monitor program, or *burning* if the file goes into an EPROM to plug into the target computer board.
6. Try out the program to see if there are areas that don't work or need improvement (*debugging*).

## DEVELOPING SOFTWARE WITH $\mu$ VISION

Previously, developers carried out this development process step by step, typing out the instructions each time to compile, to link, and to get a HEX file. Under DOS, you could simplify this with batch files. Now the process is even simpler with Windows® tools such as  $\mu$ Vision (included with this book). In my opinion, once you have tried it, you will never want to go back.<sup>4</sup> Let me outline the development process with  $\mu$ Vision:

1. Define a new project—the overall programming effort—(*Project, New Project*) that will hold all the program pieces together—pick a name that describes the overall function of the finished software.

---

<sup>3</sup>With most 8051 tools, linking, and locating are handled in one step.

<sup>4</sup>If you do want to use DOS and make your own batch files, the process is outlined in Appendix A2.

2. Open a new file (*File New*) and type in the source code for a module (which can be the entire program)—the instructions you have come up with to get the microcontroller to do what you want. Give it a name and extension, using *File Save As...*
3. Update the list of project modules (*Project, Edit Project*) to include the module in the project.
4. Click on *Update Project*. The following steps happen automatically. First, the compiler (or assembler) interprets your typed file to come up with machine code.<sup>5</sup> Next, the machine code has these specific addresses filled in.<sup>6</sup> Then the entire collection of combined segments is *located*—a process where the code has final addresses filled in.<sup>7</sup> The module is in a very condensed form, one byte to an instruction byte

---

<sup>5</sup>I assume you have set up the  $\mu$ Vision environment parameters described in the last part of this chapter. Unless you have written absolute code in assembly (using the *ORG* directive), this module will be relocatable—the jump and call addresses are not yet filled in and the pointers to variables are still undetermined. **Object files:** The object file, the condensed form of the code from one module, is the code that is yet to be linked and located. It is nonprintable, although, if you display it with  $\mu$ Vision, it is converted to a viewable HEX format. Usually an object file has a lot of additional information besides just the code. It often includes the names of variables and the various relocatable modules for the linker to use. You can have additional debugging information added to the final module so the simulator can view variables by name and show C source code for the debugging.

<sup>6</sup>If there are multiple modules, this requires *linking* so the individual segments of code are combined into one segment, the on-chip RAM segments are combined, and the off-chip RAM segments are combined.

<sup>7</sup>The process of locating is almost unavoidable with 8051 programs because they are not *load-time locatable* like programs in the x86 processors in PCs. In that processor family there are segments or selectors that add to the offset to make up the physical address, so a program can be put at a different memory address just by changing the selectors. It is technically possible to make relocatable modules for the 8051 by avoiding all instructions with absolute addresses. For starters, that eliminates *LJMP* and *AJMP* and all the calls. In addition, you would have to avoid any absolute variable references, so any variables would have to reside in a registerbank. This is treading very close to the issues of reentrant functions. In short, virtually all 8051 programs have to be located to absolute addresses before loading into the processor.

and is now a nonrelocatable, absolute object module. Finally, the software converts the resulting code into HEX notation.<sup>8</sup>

That is as far as I go automatically with  $\mu$ Vision. You can choose to download immediately to an EPROM emulator (which I do not have) or to go directly to *ds51*, but I have not wanted to move that fast. When I become more familiar with it, I may see that differently. In the pages that follow, I take you through the details of the development process as done with  $\mu$ Vision.

---

<sup>8</sup>The linker/locator will have produced a file where a byte of code is represented by a single byte in the file. In order to have an object file that can show on a CRT or be printed, this step converts the file to a form where each code is displayed as two bytes—the alphanumeric codes for the two hex digits that describe the *single* byte of code.

Since the mid 1970s, both Intel and Motorola have been expanding their respective machine-printable notations for programs, called *Intel HEX format* and *Motorola S records*. Most debugging and PROM programming tools require one of these formats, and the tools provided with this book can make the conversions.

---

```
:09000300E5902403F5B080F82219
:0300000002000CEF
:0C000C00787FE4F6D8FD75810702000340
:00000001FF
```

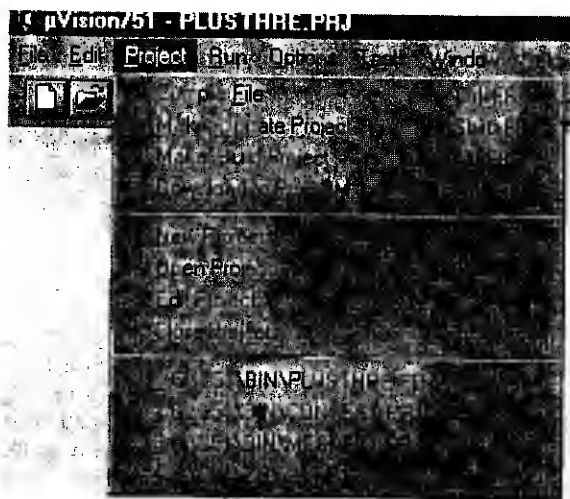
---

This file is the HEX file for the original program to add three to the switches. The colon indicates the start of a block or record. All the bytes of the file are printable ASCII codes. The first two digits (09, 03, 0C) indicate the number of actual bytes in the record. The most common value, a 10<sub>16</sub>, represents a block of 16 bytes of data. The second block has 3 bytes of code—02, 00, and 0C. Following the record length are four digits that are the location, in HEX, where the bytes should be stored—0003<sub>16</sub>, 0000<sub>16</sub>, and 000C<sub>16</sub>. Following the address are two digits that specify the type of record—00 for absolute code and 01 for an end-of-file record. Intel now defines numbers for other records such as relocatable files and files related to features of the 8086. This is an example of a simple system becoming more complex with additions undreamed of by the initial developers. Next comes the actual data where each HEX digit pair represents a byte and 16 bytes are represented by thirty-two characters. The final two digits are the checksum—the number which, when added to all the other two-digit numbers in the record, gives a 00 result. Obviously FF + 01 = 00, but a little checking confirms that 03 + 02 + 0C + EF = 00 also. When all the two-character HEX values are added up modulo 256 with the checksum, the total for an uncorrupted block is zero. Presumably any good software tool taking in these blocks would immediately recognize that things didn't "add up" and would refuse the record if there were an error. The whole area of error detecting and correcting codes is far beyond the scope of this book, but suffice it to say, a simple checksum is probably sufficient for the likelihood of error in this situation.

## Defining a Project

A project in this context is the entire software package. If you have only one module, as is likely at the start before you have gotten to modular programming, there is only one file in the project (or perhaps an initialize or startup module). As you progress, there may be any number of modules representing small parts of the overall job. In either case, you need to have a name for the overall project and the resulting absolute code module.

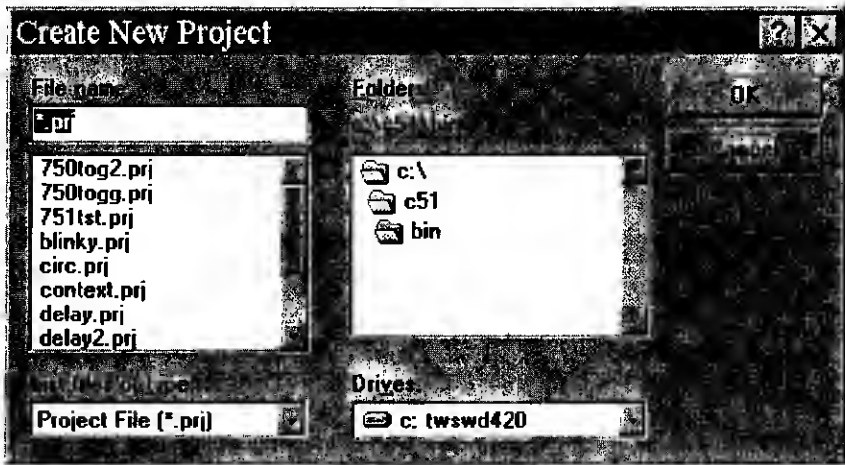
**Project files.** To start, click *Project, New Project* and give it a name. Later you will put files into the project, but for now you want a handle to attach to the collection of files and the option settings you make for processing the modules. From here on, all the new files you create will be available to include in the project.



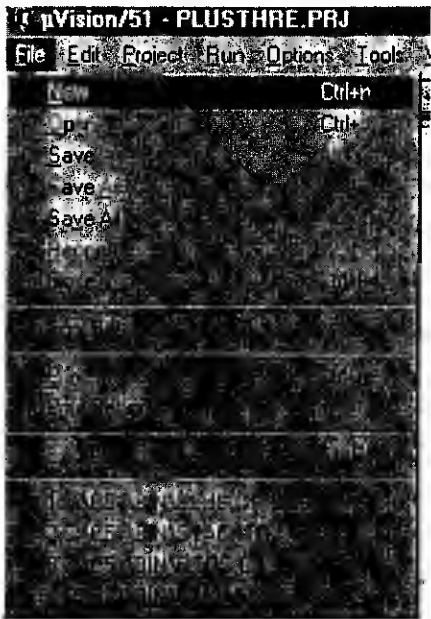
**Project setup**

**Source files: Naming.** Source files are ones you type in made up of alphanumeric codes (ASCII code<sup>9</sup>) for each character. The source files spell out the instructions in human-understandable assembly language or C. They make no direct sense to the microcontroller, but describe what you, a human, want to have happen in the computer. The process that follows typing will convert them to something the 8051 “understands.”

<sup>9</sup>Incidentally, you pronounce it “ass-key.” It stands for the American Standard Code for Information Interchange—there is no Roman numeral II in it!

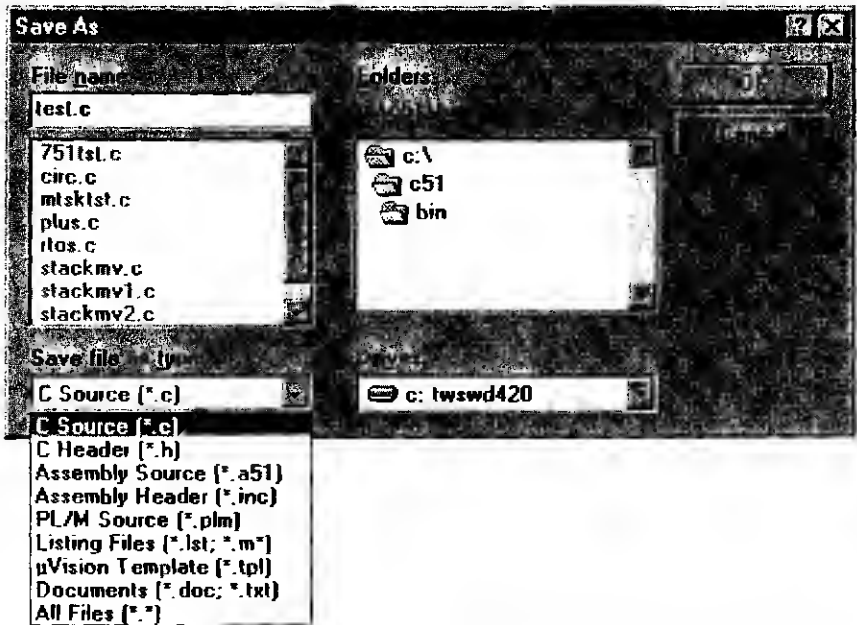


Project naming



Opening a new file

When you are ready to start typing your program into a file, *click* on *File* and drop down to *New*. That gives you a window with a caption of *<Untitled 1>* to begin entering in text. Before you forget to do it, *click File, Save As...* to attach a name to the file.



Save-as window

As you can see, you have choices as to the file name (I filled in *test*), the type of file (which determines the file extension), and the storage location [here I am keeping my intermediate and final files right in the folder (subdirectory) *bin* where all the software tools reside. This is probably not a good location—you will mix your files in with all the software tool files, but it got me by until I discovered how to specify the paths in the *make file* setup].

**A sample C program.** I discussed this same program at the start of Chapter 2. Here it is just a piece of program to work with—the details were covered before.

### Entering text.

A most useful feature of the editor is its ability to recognize C constructs and comments. Although I can't show it on a black-and-white page, my CRT screen

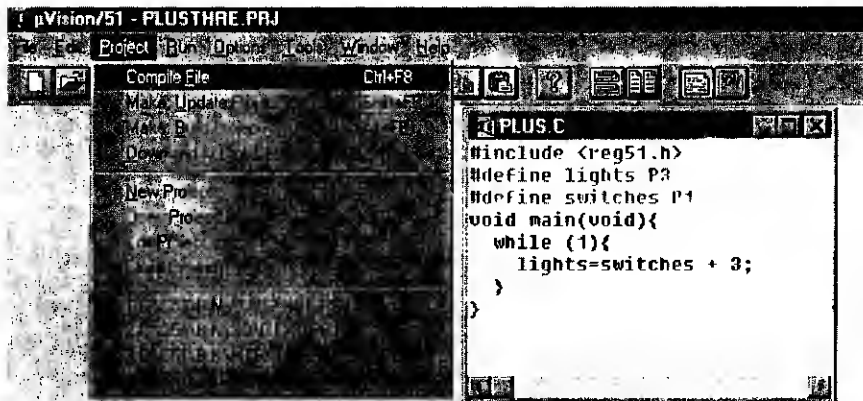
```
#include <reg51.h>
#define lights P3
#define switches P1
/*a simple lights-to-switches program*/
void main(void){
    while (1){
        lights=switches + 3;
    }
}
```

shows the *define* and *include* lines in gray, the

### My sample program

comment line in green, the C constructs (*void*, *while*) in blue, the symbol names (*main*, *lights*, *switches*) in red, and the constants in purple. If you have ever missed the end of a comment line and had a large block of program compile as text, you can have an immediate appreciation for the value of the colors. Likewise, if you misspell *while*, it will not be blue, meaning that it is not a recognized C word.<sup>10</sup>

**Just compiling.** When you are just starting out it is very likely that you will have errors. While *Build Project* or *Update Project* will stop if an error comes up, the error message seems less intimidating if you only ask to compile (*Compile File*).<sup>11</sup> The results are either a message that you have errors, with a handy way to jump back to the source line where the error appeared, or a message that it was successful.



### Compiling the program

<sup>10</sup>Unfortunately this color feature does not apply to assembly source files—it would be handy to have the editor highlight recognizable assembly instructions and comments.

<sup>11</sup>Technically assembling is not compiling, but if you've set up for an assembly language module, you get assembly instead.



**Making the project.** Getting from the source files to the HEX result is called the *make* process. Unlike the step-by-step or common batch file processes described in Appendix A2, make files can decide whether a step is necessary based on the relative ages of the files. If you have not modified a file since the last make, the environment will not run the compiler. When there are several large modules to a project, that can save a significant amount of time.

*Make: Build Project* always does the entire process from scratch. *Make: Update Project* also does it from scratch the first time, but will do only the necessary steps after that. The environment includes linking and locating even when *none* of the source modules has changed.

**Where is the program?** In the absence of any special directives, with the settings outlined you get a HEX file [*myfile.HEX*] which can be used by a PROM programmer and [*myfile.M51*] which shows specifically where the various variables and code modules are located. By default the linker/locator has the code start up at address 0000<sup>12</sup> with off-chip RAM (*xdata*) also beginning at 0000. This is useful for final ROM-based systems such as with the 87C51, but it does not fit with downloading a monitor program as with the MCB520 or PU552 boards.<sup>13</sup>

If you choose to write in assembly for downloading with a monitor, either make relocatable modules (the *RSEG* directive) and direct the linker/locator, or use *ORG* to force the code where you want it.<sup>14</sup>

## INSTALLING $\mu$ VISION

Having described how you use the Keil software tools, the pages that follow take you through the process of installing and initializing the software. You

---

<sup>12</sup>All the 8051 family start up on power up at 0000. By comparison, the 8086 family starts up at FFFF0<sub>16</sub>.

<sup>13</sup>With those boards, off-chip data (*xdata*) and off-chip program (code) addresses point to the same chip (necessary to download code into RAM for development) so, if your program uses off-chip RAM for variables, the RAM (*xdata*) addresses cannot overlap the program (*code*) addresses. The process is tricky—the easiest way to get your code to stay out of forbidden areas is to link in a dummy module first that occupies the forbidden space, leaving only the “good” space for your code and variables. The monitors are described later in this chapter and the two boards are described in Appendix A4.

<sup>14</sup>If you decide to use *ORG* with multiple modules you take on the job of the linker and must be sure the code pieces do not overlap. Even more challenging is ensuring that the variables do not overlap—particularly as modules change in the debugging process. I recommend *ORG* only for single-module programs.

need a recent PC (at least a 486), several megabytes of free disk space and 4M or 8M of RAM, depending on the version of Windows® you have. If you are already running common Windows® applications, this package will seem quite small by comparison. As of this writing there is still a version of Keil's tools that runs under DOS, but Windows® is so prevalent I omit any discussion of it.

The installation process is similar to most other Windows software:

1. Insert the disk.
2. In the *File Manager* (or *Windows Explorer* in WIN95®) click on the *D* drive (or whatever letter applies to your CD-ROM) and then click *Setup.exe*.
3. If you have no reason to do otherwise, accept the default settings (*C* drive, *C51* folder).
4. When the setup is done you should have an icon for *µVision*.<sup>15</sup>

### Setting Parameters

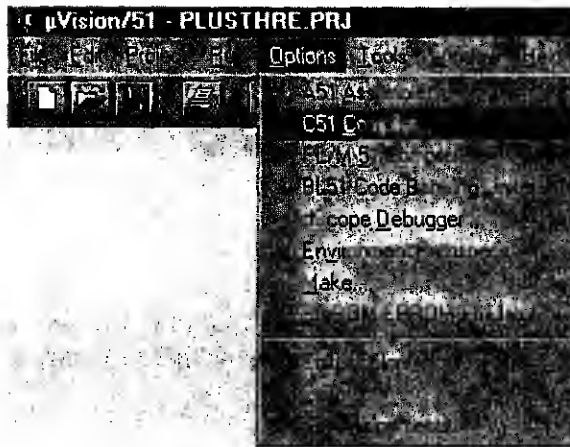
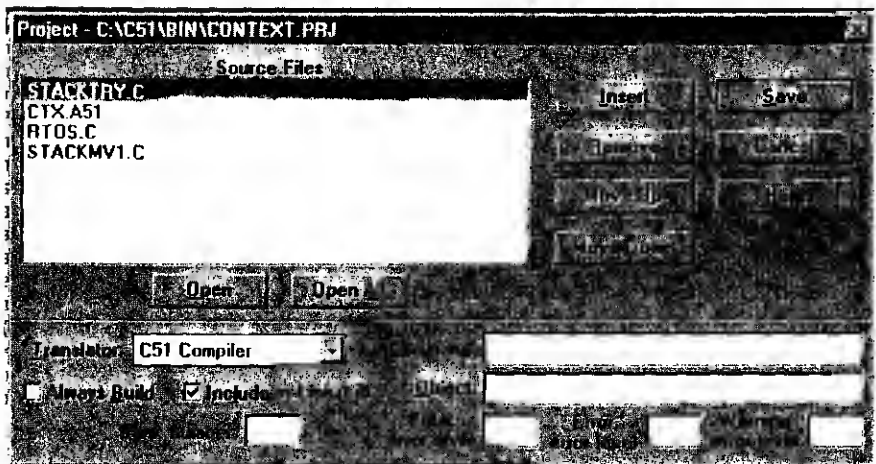
The pages that follow show the various settings you can make as part of the configuration process. To begin with, click *Options* to see the various aspects of configuration.

**Project file again.** The project file is a listing of all the modules that make up the total program along with the options you have chosen for the various steps in the process. Of particular interest is the sequence in the source file list. The linking happens in the same sequence unless you override it.<sup>16</sup>

---

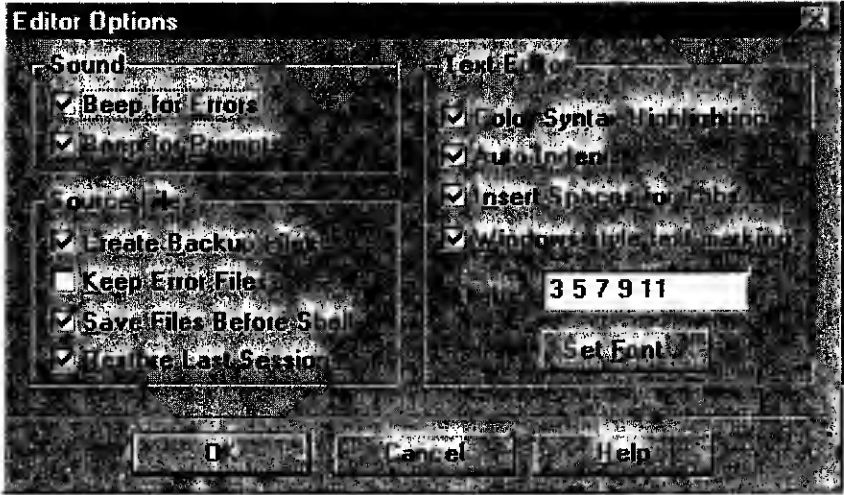
<sup>15</sup>You can, depending on your PC operating system, drag the icon to a window of your preference with Windows 3.1 or set a shortcut to it in WIN95® to put on the desktop or in a sidebar. This is an operating system issue I will assume you learn(ed) elsewhere.

<sup>16</sup>The linker includes the various modules *in the order it encounters them*. When developing for nonstandard memory configurations (such as with a monitor for downloading) be sure the *first* file in the list is a module that claims the space the monitor uses. That will force the code to be at different addresses than the monitor firmware (code in ROM). It will also keep the data at different addresses than the downloadable code, since they both get put into RAM space. If you reverse the order, the start address will move around depending on the size of the program and the code will be assigned the space where the monitor is located. The default locations put the code just above the interrupt vectors with a long jump at 0000. You cannot download there if the space holds an EPROM and, if it holds RAM, you have no startup program to do the downloading. For more on this, see the section on development tools.

Picking the Options menu

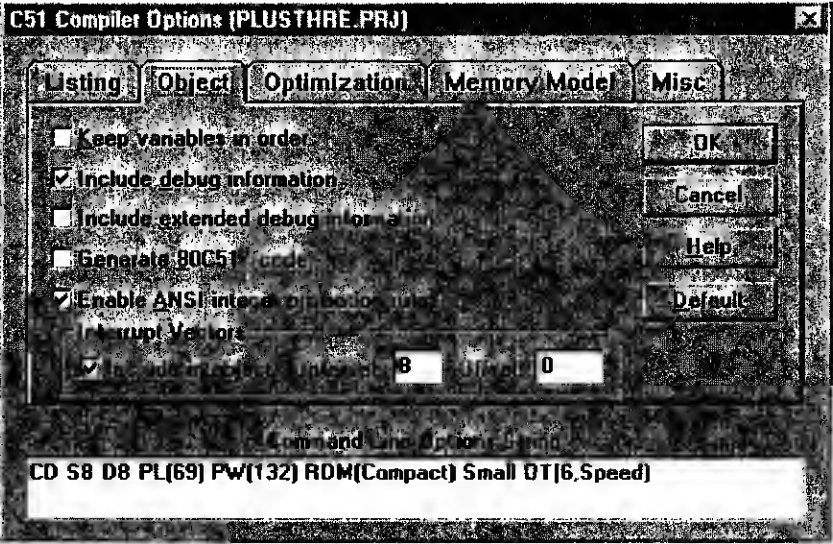
Project example

**Editor features.** Click *Editor* to see the options screen. Most of the choices I've shown are optional. I recommend especially the *Color Syntax Highlighting* which I described earlier, the *Auto-Indent* which starts the next new line at the same indentation as the line above it, and *Insert Spaces for Tabs* which keeps your listing consistent across different printers where the tab settings may differ. You can play with the other features. As I recommended in the pages on program style, I prefer to write C with shallow indents (every 2 shown here) to save room to the right for comments.



Editor options screen

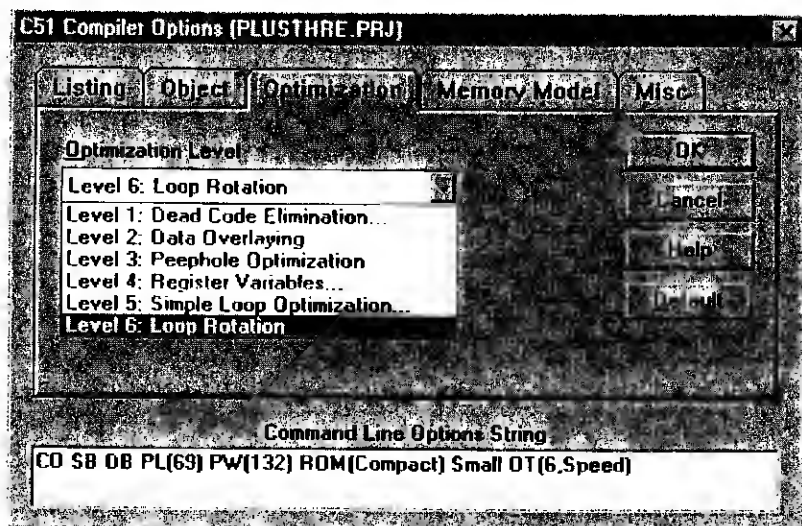
**Compiler settings.** Again under the *Options* menu, click *C51 Compiler*. Then pick the *Object* tab.



Compiler options—object files

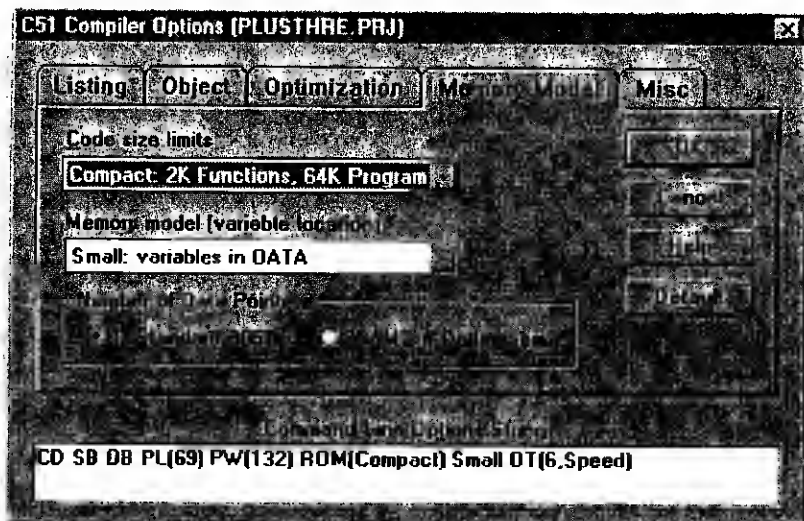
Some of these features seem unimportant. The lower few have to do with a few family members that can pack in the interrupt vectors more

tightly than the default for the 8051. The only feature I use is the debug information. It must be checked if *DS51* is to get symbol names and C source lines for viewing—even a powerful computer cannot regenerate your symbol names and comments from raw machine code!



### Compiler options—Optimization

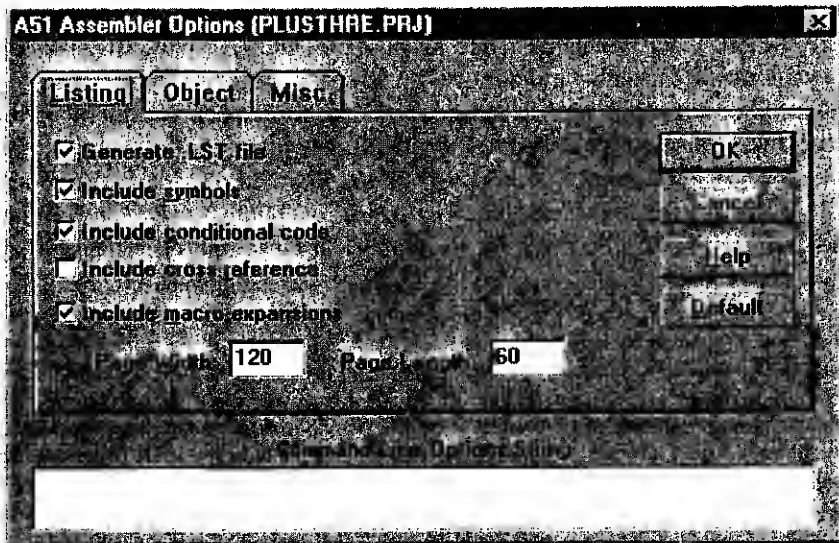
Click the *Optimization* tab. You need the compiler manuals if you are going to make this a matter of choice. Otherwise, pick the highest level.



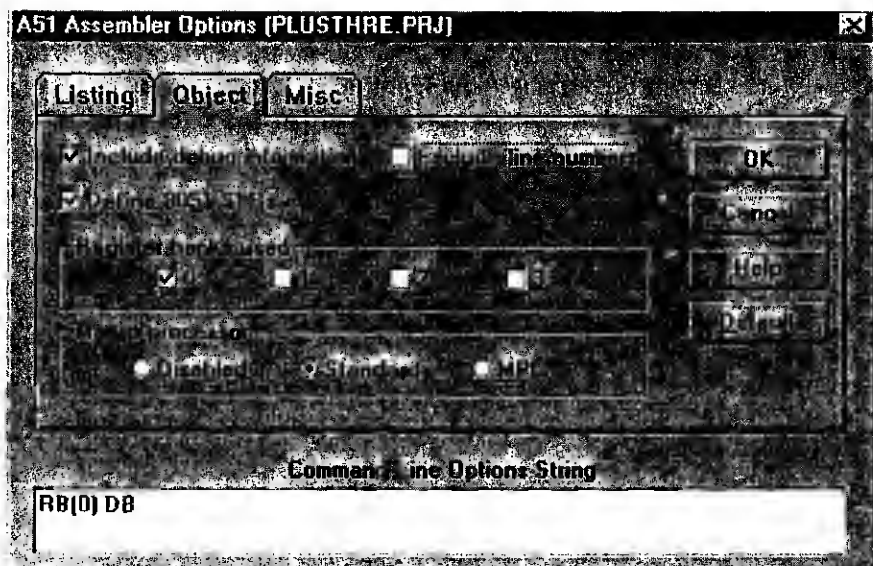
### Compiler options—Memory model

Click the *Memory Model* tab. Here you determine what the compiler does with variables when you do not tell it otherwise. The setting above is the one to keep all the variables on-chip. The code setting forces the compiler to use *AJMP* in all the functions (but *LCALL* to get to them) since each function is to be under 2K in size. The *Small* setting [*ROM(Small)*], necessary for the 87C750 and 87C751, also avoids all use of *LCALL* because it is not supported and is unnecessary in chips where the total code can never be larger than 2K.

**Assembler settings.** Click *Options A51: Assembler* and then click the *Listings* tab. This is straightforward. If you use conditional code or macros, you are an advanced programmer who needs the manuals. I think a page *Width* of 120 is a poor choice unless you have a wide carriage dot matrix printer. I suggest 80—this setting is never a problem for me because I do not write wide assembly programs.



Assembler options—Listing



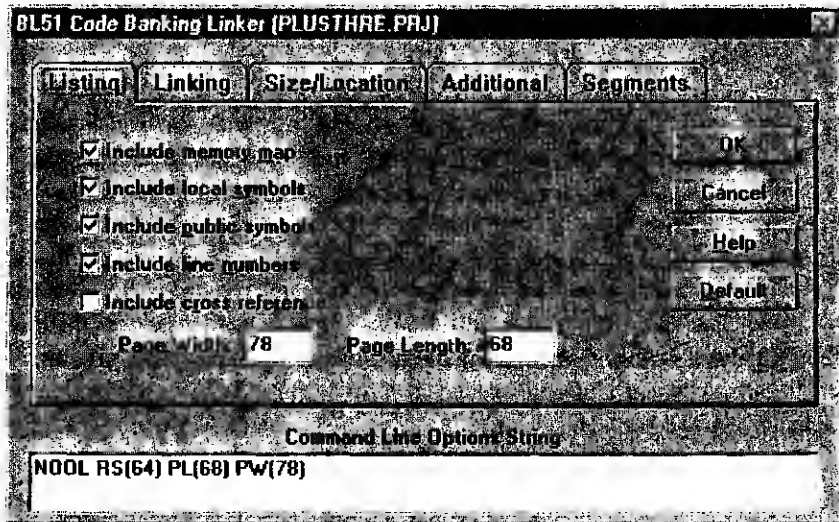
### Assembler options—Object

Click the *Object* tab and *Include debug information*. Unless you want to define all your own register set or use one of the other assembly *include* files for a different family member, let the assembler automatically recognize the 8051 register names (like *PSW*, *ACC*, and *IE*). Note that you can specify that a particular module uses a specified register bank. When you get into interrupts you may have to override this for specific segments, but I expect you will have switched to *C* by that time.<sup>17</sup>

**Link settings.** Click *Options*, *BL51 Code Banking Linker* and then the *Listing* tab. You do not have to include all these options when you are

<sup>17</sup>Simple assemblers are supposedly easy to write—at least when only simple conversion of character strings into machine codes are involved. Many of the C compiler suppliers include assemblers as well. Certain compilers produce only assembly mnemonics (assembly language), which you must then, as another step, assemble to get machine code. If you intend to mix languages, do not assume that one company's assembler will mesh with another one's compiler. Keil/Franklin's languages seem to mesh well with Intel's languages and tools. BSO/Tasking's tools are quite different and you must use them as an entire package. In-line assembly code is an option with certain compilers.

done, but it is handy to see where the linker stored the variables. Line numbers are only useful if you are debugging C code in a monitor.<sup>18</sup> Cross-references are useful when you combine many separate modules.



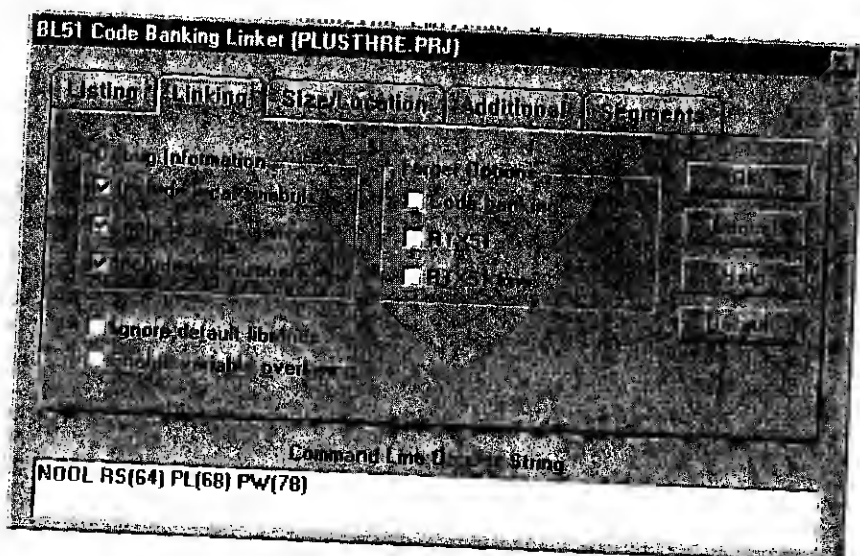
### Linker options—Listing

Click the *Linking* tab. The debug information is crucial if you want to use the simulator (DS51). I assume you are not using code banking or the real-time operating system that only comes with the professional software version. Variable overlaying is useful if you have several functions you call separately that use temporary variables.<sup>19</sup>

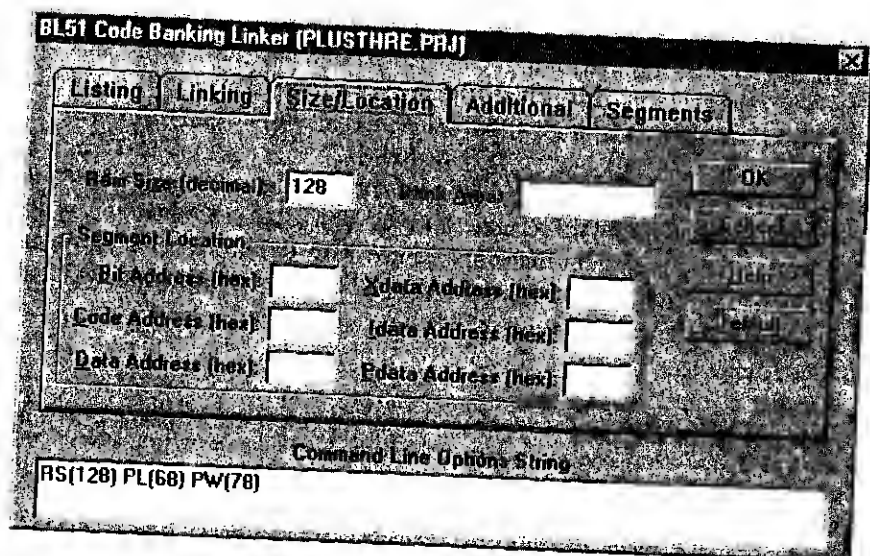
<sup>18</sup>When debugging with a monitor you have to translate C program lines to the actual addresses where the corresponding machine code starts. If you are going to troubleshoot to *within* a line of C, you would have to direct the compiler to include the (assembly) code listing in the *.LST* file. If you are using the simulator, it is easy to display in the mixed mode (C and then the corresponding assembly on a line-by-line basis) so you do not have to use the line numbers.

<sup>19</sup>You should declare *static* any variables within a function that need to remain between calls. Never assume an *automatic* variable holds its value until you get back to the function. You may get away with it sometimes because a simple program may not have multiple, non-overlapping functions to overlay, but it will probably catch you when you start building up to larger programs.





Linker options—Linking



Linker options—Size/Location

Click the *Size/Location* tab. The essential setting is the size of on-chip (*data* and *idata*) RAM. The 128 is for the 8051s. The 8052s and most new

chips have 256 and the 750/1 has 64.<sup>20</sup> The *Segment Location* settings may eliminate the problem of forcing downloadable code locations by setting the *code* address to begin, say, at  $2000_{16}$  and *xdata* address to begin at  $3000_{16}$ . Unfortunately the process may not be that easy—you may have to enter all the segments *by name* (a dialog box shows up to help you), and the steps must be done each time you put together a new project. The dummy module approach is easier and less work for a changing variety of programs.<sup>21</sup>

**Make files.** The make files describe the steps to get from the source code to the final machine code. It is the process of *making* the final code from the source pieces. The environment does this so easily you may not appreciate it—students ought to have to do the steps one-by-one for a while to appreciate the difference. Looking under the project menu you can see the options of updating or building the project. Building the project means doing all the steps from scratch—compiling or assembling *every* module before linking. Updating, on the other hand, only compiles if the source has changed since the last time. With a large multimodule program, the difference is significant—no need to recompile a module that has not changed



**Make options**

<sup>20</sup>The most unusual was the 8044 (no longer in production, used with BITBUS) that had 196 bytes.

<sup>21</sup>Appendix A2, page 382, shows a dummy module.

since the last time. By checking the date and time for each file, the software determines if the object file is older than the source and only recompiles if needed.<sup>22</sup>

Click on *Options Make* and select the *After Compile* tab. The conversion to HEX file formats is necessary to work with many PROM programmers and some debuggers. I discussed the use of the librarian in Chapter 8.



**More Make options**

Click on the *After Make* tab. If you are heavily into simulation you can have the environment jump you right into the simulator.<sup>23</sup> For now, I suggest you just stop. The other choice is another commercial tool.

<sup>22</sup>I would have assumed that if the link file is no older than any of the object files it would omit linking as well but that does not seem to be the case—it always re-links everything.

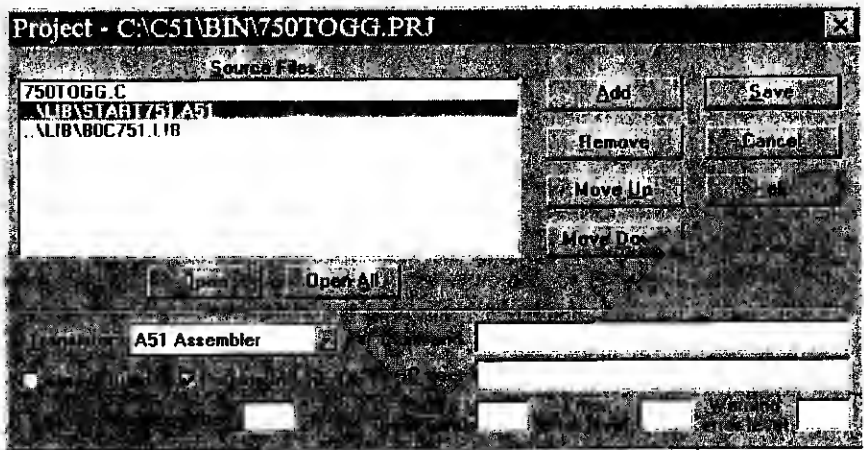
<sup>23</sup>In the simulator, for automatic entry of a special program and environment, you will have to make up a startup file that is invoked when *DS51* is entered. That level of preparation is very useful when you're working on one project for weeks or months but isn't worth the effort at the start.



Make options

Click on the *Extensions* tab. The only change I can see in the default would be if you chose to name all your C files with a *.C51* extension to differentiate from C files on the host PC.

**Project files for the 751.** The 750/1 family members are limited to 2K of code (non-expandable) and do not allow the long jumps and long calls. Specifically with the 751, there can be no *long* jumps (*LJMP*) or *long* calls (*LCALL*) and no *MOVX* instructions. This is accomplished in C by inserting *#pragma ROM(small)* in the source file or *ROM(small)* in the invocation line.



Project file for a 87C751 target processor

You also have to override the default startup that clears out the variables before the main routine starts. Just using the small model is not enough. The previous figure shows the way to get things changed. The library, of course, omits the long calls, but the startup comes in assembler and you must assemble it before use.<sup>24</sup>

If you are not sure if the process has worked, the best check is to look at the HEX file for the jump from zero.<sup>25</sup> If the jump is a long one (02H), the program is wrong. If it is a short jump (01H) then probably the rest of the program is also correct.

---

**;Long and short jumps in HEX files**

---

:02000000**01**2DD0

:03000000**02**0004F7

---

## DEVELOPMENT TOOLS

Somewhere in the development process, you have to get from software to real hardware if your work is to progress beyond an intellectual exercise. Things never work perfectly the first time, so you face the process of *debugging*. You *can* just wire up some chips and burn your program into EPROM, but I recommend avoiding the *burn and try* method as long as possible. Instead, use some of the development tools designed to ease the debugging process.

The *simulator* that comes with this book (DS51) is a good first debugging step if you have many software algorithms. You can test your program with no hardware at all to make sure that math operations work as you intended, that program loops stop after the correct number of cycles, and to generally see that the various branches behave. A simulator becomes less useful when there is a lot of timing-critical hardware. It is possible to add special programs to the simulator to model the times and responses of the external hardware, but that is a project in itself.

The least expensive way to debug hardware-related code is to *download* to a target board using a *monitor*. The download process is fast and the downloaded software runs in real time with real hardware. The monitor software allows breakpoints so you can stop to see what has happened along the way. The program code has to be located at a different address so it doesn't

---

<sup>24</sup>Again, when you include the assembly source with the .A51 extension in the project as shown above, the environment will automatically assemble it during the build process!

<sup>25</sup>All 8051 programs start from 0000 on power up. Since the interrupt vectors come in the bottom code space, every complete, stand-alone program starts with a jump over the interrupt vectors.

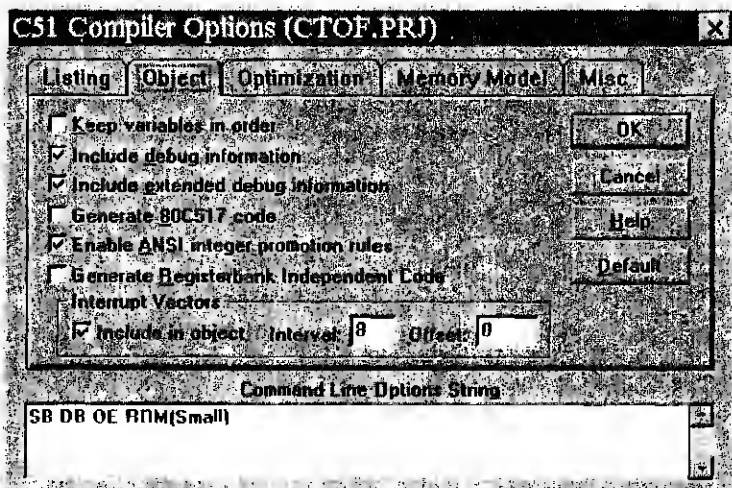
clash with the monitor, but the biggest drawback is the need for the monitor software and the tying up of a serial port for the downloading. If the port is needed in the final product, the solution is to add a second port or share the functions—a bit messy in either case.

A third development tool that has existed for at least fifteen years is the *in-circuit emulator*. Intel developed it in the 8080 days, and several companies have them for the 8051 family (see the list of addresses in Appendix A5). The emulator combines the monitor/simulator functions with the ability to use none of the target processor's resources—you don't give up program space or ports and the emulator plugs into the socket where the final processor would reside. The program runs at speed, usually, and still allows breakpoints. It works with real hardware. The big drawback is the cost—easily \$1000 or more.

The following pages describe a few of these tools in basic terms. For *detailed* information, the various suppliers are your best source.

### Simulators: DS51

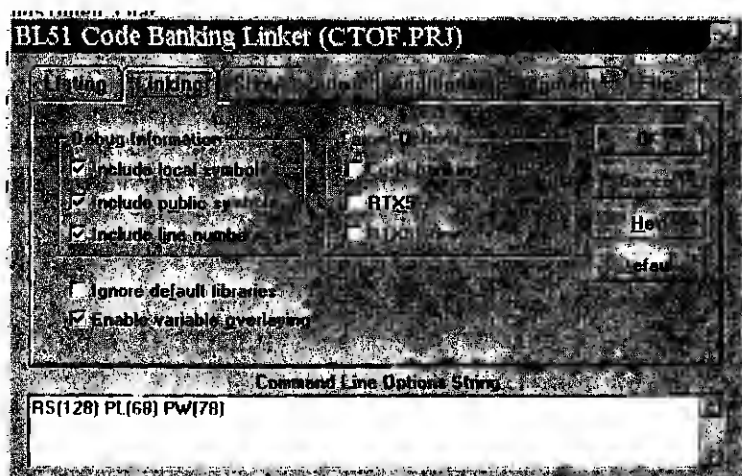
Keil includes their simulator with their other software. The simulator is an excellent tool for debugging software algorithms and for initial debugging of the general program flow. It is much more difficult to use if you have to respond to external hardware—particularly interrupts.



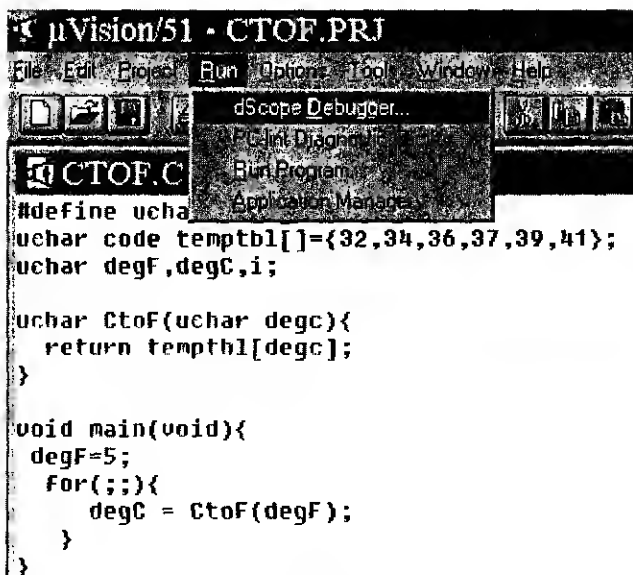
### Setting compiler options to include debug information

If you are going to use the simulator you need to change the options for both the compiler and linker to include the symbols and the actual source code. The two screen captures that follow show these settings.

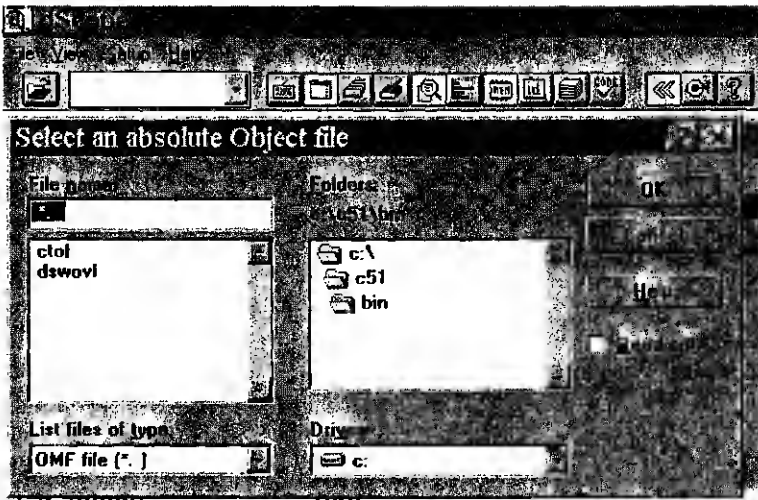
If you start dScope from  $\mu$ Vision, it is one of the run options. You can also set the *make* options to flow directly into the debugger after any recompiling, but I find it keeps giving new invocations of dScope—easier to just move back to the original invocation and reload the file.



Setting linker options to keep symbols



Invoking dScope



Picking Object file

Once you have entered dScope, you can click and drag in the usual Windows® way to rearrange the various views to please yourself. I show an arrangement that satisfied me for debugging the *CtoF* program from an earlier chapter.



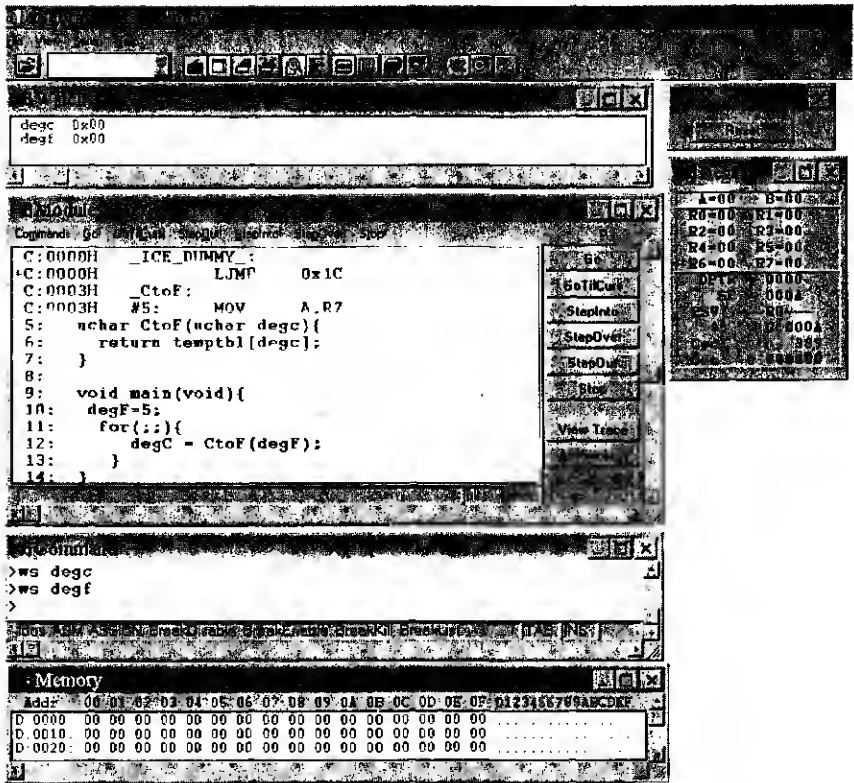
Loading object

Once the file is loaded you will probably see the C source code (*View High-level*). If you wish, you can switch to *View Mixed* to see the lines of C with the resulting assembly for each line (it is under *Commands* for the module window—not the overall *View* menu that chooses which windows to show). If you get only assembly code, you probably missed some of the debug settings when you compiled or linked.



Picking View mode

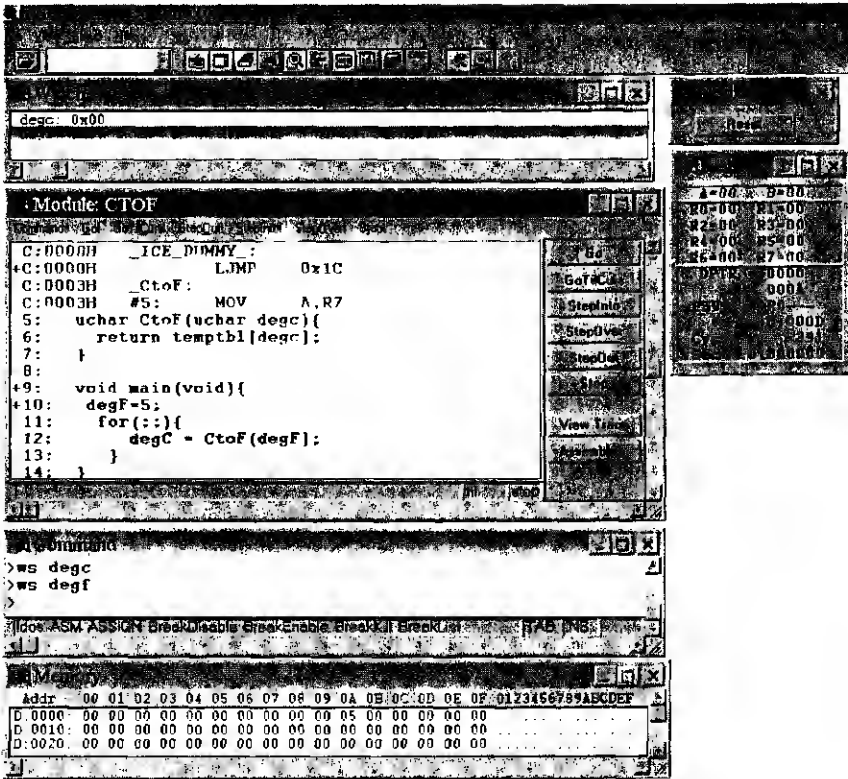




Screen at start of CtoF program

Of course, if you are debugging an assembly program, that is a different matter!

The first screen here shows the start of the *CtoF* program from Chapter 6, page 133. I have set a watch on the *degc* and *degf* variables (>WS *degc*, and so on, in the command window) and have arranged the memory and register windows to fit on the screen. From *reset* (click on it in the toolbox window), I set the cursor on line 10 and click *GoTilCurs*. That brings the simulation through all the initialization as you can see, and everything is zeroed (and it took 389 machine cycles).



### Screen stopped at line 12

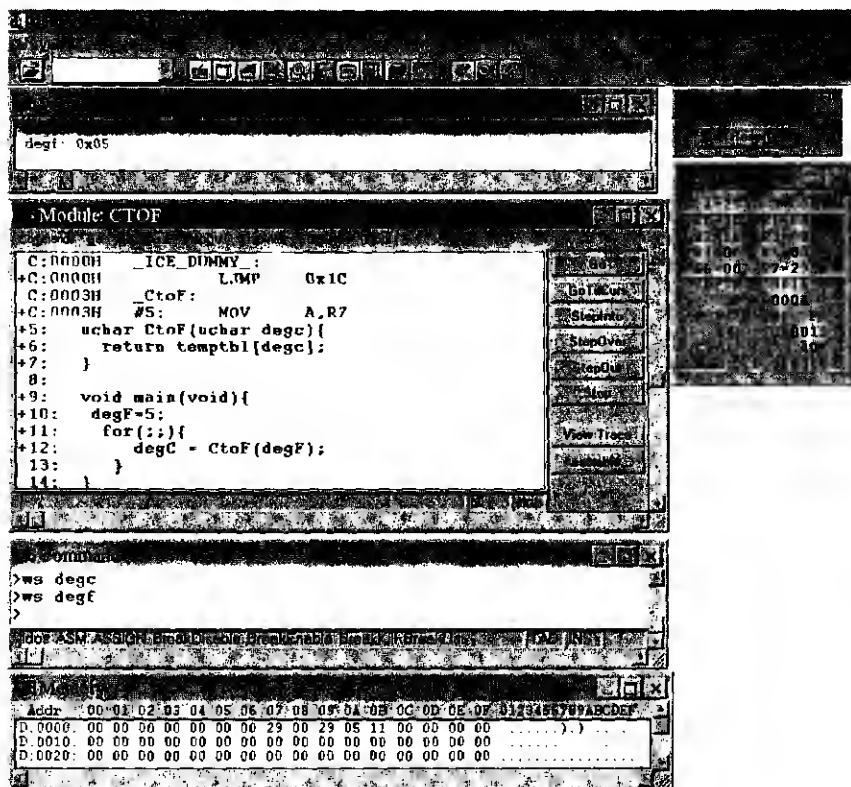
The second screen capture, stopped with the cursor at line 12, shows that the program has now filled *degF* (at  $0A_{16}$  in memory) with 5. The program has not yet executed line 12 where the cursor is located.

The final screen shows halting after the function has returned its value. Incidentally, I have just discovered an error (which I went back and fixed in Chapter 6). I named the incoming variable *degF* and the result *degC*, but a quick check of the simulator results shows that I reversed the names.  $5^{\circ}\text{F}$  is not  $29_{16}^{\circ}\text{C}$ , but rather the reverse! The simulator does catch many errors in thinking.

Notice that the result, *degC*, is in memory location  $09_{16}$ . Location  $07_{16}$  has the result of the function call.<sup>26</sup> I can quickly see that the cycle count is  $405_{10}$ , so the entire function took  $16_{10}$  machine cycles to execute. If I had wanted to see the function details, I could have put the cursor at line 6 or

<sup>26</sup>This agrees with the table on page 191 of Chapter 8 that says a byte returns in *R7*.

else clicked *StepInto*. *StepOver* treats functions as single lines. There are many more details to using the simulator, but this should get you started.



Screen after `CtoF` function has run

## Monitors

You can do most development work using the target microcontroller as the debugging tool. By installing a firmware monitor program (EPROM) which starts on power-up, you can communicate with a PC that also compiles your code.<sup>27</sup> The PC interfaces to the target system over a serial link and the monitor program allows you to download new code attempts to

<sup>27</sup>Two monitor programs are described in the following pages and two target boards are described in Appendix A4.

RAM on the target system. You can then run your code with single stepping or break points. When your program stops, the monitor program resumes running and allows you to examine data and code memory and make changes.

The main *drawbacks* of this method are:

1. Your hardware must include an extra serial port.
2. Logic must OR PSEN and WR so code and data space overlap to allow the monitor program to write to code space for downloading.
3. You must locate your code away from the 0000 startup location. This is not difficult, but you must relocate the *final* burned-in code and you may have to make a few changes in the interrupt vectors.

The main *advantages* of a monitor for development are:

1. A monitor is definitely more economical than an emulator.
2. You can run your program at real speed (unlike a simulator).
3. Your program interacts with real hardware.
4. You can use breakpoints (unlike burn and try).

**How a monitor works.** A monitor itself is just a program running on the target computer.<sup>28</sup> Characters typed on the keyboard of the host terminal come in through the serial port. The monitor responds to these command characters by sending the appropriate message characters back to the host computer's screen.

When your command requires an address, the monitor saves it in a reserved area of memory.<sup>29</sup> If you display the contents of the registers, a mirror image of the run time contents must have been stored because the EET monitor needs to use the same registers when running.<sup>30</sup> When your pro-

---

<sup>28</sup>Note that any of these monitors take up target computer resources—at least the code space—and require the target system to include off-chip RAM/code space for the downloaded program and at least one serial port. For simple, single-chip applications you must either use an emulator (see section on development) or develop on a development board and then make the necessary code changes before burning the final program into the single-chip version.

<sup>29</sup>This area for ETMON18 is between  $3F00_{16}$  and  $3FFF_{16}$  in external data space.

<sup>30</sup>The monitor stores these registers in internal RAM from  $5E_{16}$  to  $7E_{16}$ .

gram is about to run, the monitor reloads the registers with their run-time values. At a breakpoint or a return to monitor, it again saves the registers so that you can examine or modify them.<sup>31</sup>

To test interrupt-based software, the monitor program must be able to let your code respond to hardware interrupts. Each interrupt forces a call to a memory location between  $0003_{16}$  and  $0023_{16}$  in code space. Since this ROM space is occupied by the EET monitor, there must be JMP instructions at these interrupt vector locations to send your program to the download code/xdata RAM. With the EET monitor the RAM vector locations are  $3F00_{16}$  above the interrupt vector addresses. You have to get the interrupt code to the new vector jump location. As an example, consider a case using interrupt 0. When the interrupt occurs, the program flow vectors to  $0003_{16}$  to service the interrupt. At that location the EET monitor has a jump to  $3F03_{16}$  ( $3F00_{16} + 0003_{16}$ ). Your downloaded program needs a jump at  $3F03_{16}$  to vector to your actual interrupt service routine.

The GO command simply executes a JMP ( $02_{16}$ ) to the start address of your program and the micro begins running code there. From here on the monitor has no control at all over the micro. If a *breakpoint* has been entered as part of the GO command before starting your program, the monitor program first saves the next three instruction-bytes after the break address, and replaces them with a 3-byte *jump-to-monitor* command. When (or if) your program reaches the breakpoint address, the jump forces program flow back to the monitor. The monitor program then restores the original code that it replaced by the jump.<sup>32</sup>

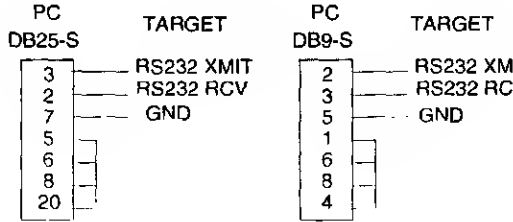
---

<sup>31</sup>The running program breaks back to the monitor because the latter has temporarily substituted a jump at the breakpoint. At the end of subroutines (for downloaded code running in RAM) a monitor can add a breakpoint only by substituting a 3-byte LJMP for another instruction. If it tries putting a 3-byte instruction in place of a RET at the end of a subroutine, it will overwrite the first two bytes of the *next* subroutine. If the program uses the second subroutine before it reaches the breakpoint, the result is disaster! If you put two NOPs between subroutines, this is not a problem. This is not feasible in C, so, when debugging with a monitor program, be careful about putting breakpoints near the end of functions.

<sup>32</sup>The break-out capability is only possible because the code is located in alterable RAM—not the case except for code downloaded into a jointly code/data addressed memory space. If the breakpoint is not reached, the original code is not restored and you will have to stop the program with the reser and then download again to get a “clean” copy of your program into the target micro.

## EET MONITOR PROGRAM V1.8 (EET Monitor) for 8051 Family Microcontrollers<sup>33</sup>

**Getting started.** Insert the ROM containing the EET monitor program into the socket on the target system board (on the PU552 it comes pre-installed, but you can use it on other boards). It must be in a socket mapped into external code space starting at address  $0000_{16}$ .<sup>34</sup>



You can interact with the monitor from a PC running a terminal-emulation program<sup>35</sup> that allows serial communication as well as file trans-

<sup>33</sup>This section provided courtesy of Richard H. Barnett and Neal S. Widmer. The EET monitor was written in 1991 by Dr. R. H. Barnett to provide students with an appropriate development system for the Taber Microprocessor Lab, funded by a grant from the Hoffer Plastics Foundation. The source code is written in assembly language and may be assembled by Intel ASM51 or Keil/Franklin A51. It has been updated several times and is currently capable of dealing with Intel 8031, 8032, 8051, and 8052 processors, as well as the Signetics 80C552. The software and this documentation may only be used for noncommercial purposes and you may reproduce it to give away but not to sell. You may only make copies of the monitor program and documentation in unmodified and complete form. The source file name is ETMON18.A51.

<sup>34</sup>The EET monitor assumes that the target system has a Read/Write memory starting at address  $2000_{16}$  that is mapped into both *code* and *xdata* space. This is necessary for downloading programs into *xdata* and running them as though they were in *code*. If a program is located at an address less than  $2000_{16}$ , the monitor will replace the upper byte of the start address with  $3F_{16}$  and load the code at  $3Fxx_{16}$ . This was done to facilitate use of PLM-51 code (which locates RESET and interrupt vectors at  $0000_{16}$  to  $0025_{16}$ ) to be downloaded and tested using the monitor. Note that any jumps to absolute addresses within the program code below  $2000_{16}$  would not work when downloaded and relocated.

<sup>35</sup>Terminal emulation programs include PROCOMM and SEND31 (refer to SEND31 section), but are often provided with other software.

fer from a floppy disk. The target system, running the monitor, receives serial data from the PC keyboard and responds by sending serial messages to the PC screen.

You need an RS232 cable to connect to a PC from your target system (the extra jumpers at the PC end are required).<sup>36</sup> When power is applied or your target system is reset, the terminal should beep and produce a *PU>* prompt in the upper left corner of the terminal screen. At this point, the EET monitor is ready to receive commands from the terminal keyboard.

**Conventions and details.** Throughout the EET monitor coverage that follows, the entries that you actually type are in *italics*. They are shown in upper case, but you can use lower case if you wish. **Any numerical entry is assumed to be in base 16—HEX.** The H suffix or 0x prefix cannot be used. Decimal entry is not possible.

Correct errors in typing by:

1. Using backspace to back up and retype any numerical entry
2. Keeping typing: All numerical entries will accept the last digits typed and the others will fall off the top. For example, a four digit address will take the last four digits typed. If you make a mistake at any point, type the next four digits correctly and the earlier typing will be lost.
3. Using Esc to abort the current command.

**EET monitor commands.** The monitor uses single-letter commands. These commands allow you to display or change internal registers, memory, or I/O. After many commands you choose the appropriate memory space.

#### Memory space designations

<b>C</b>	code memory space
<b>X</b>	external memory space
<b>I</b>	internal memory space
<b>R</b>	registers
<b>S</b>	stack

---

<sup>36</sup>The communication protocol is 9600 baud, 8 data bits, no parity, and one-stop bit.

## Commands

Command	Function	Example(s)
<i>D</i>	display contents	<i>PU&gt;DX 2000,2010&lt;cr&gt;</i>
<i>S</i>	substitute contents	<i>PU&gt;SI45&lt;cr&gt;</i> <i>45=00 25,</i> <i>46=FE &lt;cr&gt;</i>
<i>I</i>	input from I/O port	<i>PU&gt;II 80&lt;cr&gt;</i>
<i>O</i>	output to I/O port	<i>PU&gt;OI 80&lt;cr&gt;</i>
<i>G</i>	go run program	<i>PU&gt;G2000,2010&lt;cr&gt;</i> <i>PU&gt;G,202F&lt;cr&gt;</i>
<i>C</i>	continue running program from current breakpoint	<i>PU&gt;C202f&lt;cr&gt;</i>
<i>L</i>	loop: repeat current loop	<i>PU&gt;L&lt;cr&gt;</i>
<i>H</i>	help: display help menu	<i>PU&gt;H&lt;cr&gt;</i>
<i>:</i>	download a line of Intel HEX file	

### Commands to Display and Change Memory, Registers, and I/O

**Display command.** After entering the D command and the memory space letter, enter the starting address. Then can press carriage return or “,”. In the former case a 16-byte block will be displayed, starting at the address you entered. In the latter case, you are prompted for an ending address. The monitor always displays contents of memory in multiples of 16 bytes. The following example displays addresses 2000<sub>16</sub> through 2010<sub>16</sub> in external memory. (You enter the italicized characters and the remaining portion comes from the EET monitor program.)

---

Display eXternal memory starting at **2000H**, and ending at **2010H**

---

The resulting screen should look like this (these data values are provided as an example):

---

```

2000  30 31 32 33 34 35 36 37  41 42 43 44 45 46 47
48 01234567 ABCDEFGH
2010  00 00 3F 00 00 00 00 00  00 00 00 00 00 00 00
00  ..?.....

```

---

Notice that the starting address (base 16) is on the left, followed by the contents of that address and the following fifteen locations. On the far right is the ASCII equivalent of the number at each memory location so you can



read any text messages that may be stored in memory. If it is not a printable ASCII character, a "." holds its place.

*Display Registers* lets you examine the special function registers. It automatically senses the type of processor being used and displays the appropriate registers.

*Display Internal* memory at addresses from  $80_{16}$  to  $FF_{16}$  shows the *indirectly* addressable RAM (not present in 8051). It does *not* display the SFRs, which have direct addresses  $80_{16}$  to  $FF_{16}$ .

*Display Stack* generates a screen that shows a block of sixteen internal RAM locations that represent the stack. An arrow indicates the current stack pointer (SP).

**Substitute command.** This command allows you to change any memory location or register. After entering the *S* command and specifying the memory space, or register, you want to modify, you see the current contents. Enter a new value. *e* ends the command or , displays the next memory location or register to make it easier to enter large amounts of code or data.

For example, to substitute the values  $11_{16}$ ,  $22_{16}$ ,  $33_{16}$  into internal memory locations  $45_{16}$ ,  $46_{16}$ , and  $47_{16}$ :

---

**Substitute Internal starting at 45 Enter**

45=00 11 , (note: the 11 will overwrite the 00 in  
memory)

46=00 22 ,

47=00 33 Enter

---

You could use the Display command to check that the new numbers are actually there.

**I/O commands.** These commands (*Input* and *Output*) allow the user to read and alter (respectively) the current value on an external or internal I/O port. They are useful for internal I/O. With external I/O the Display and Substitute commands do the same thing.

## Running and Debugging Programs

**Go . . . execute a program.** The *Go* command lets you start a program stored in memory. This command loads the *Program Counter* with the specified address and turns the CPU loose running the program stored in memory.

For example:

---

**Go** execute the program at **2000 Enter**.

---

If you want to go only partway through the program, the **Go** command allows you to stop at any program instruction by setting a breakpoint. This instruction jumps the CPU back into the EET monitor, allowing registers to be examined, etc. Specify a breakpoint address by typing a **,** after entering a start address for the **Go** or **Continue** command. To set a breakpoint at address  $2014_{16}$ :

---

**Go** execute program at **2000**, and break execution at **2014 enter**

---

**Continue execution from current breakpoint.** Once a breakpoint is encountered you may wish to continue running the program from the current breakpoint (instead of starting over). The **Continue** command lets you to do this without retyping the start address. You can specify another breakpoint by using the **,** just like the **Go** command or continue indefinitely by pressing **e** (enter).

**Loop . . . execute current loop and stop at current breakpoint.** You may want to stop a program each time it goes through a loop to see if each iteration is producing the desired results. After reaching a breakpoint, the **Loop** command resumes the program at the current address and breaks execution the *next* time it reaches *this* address (in other words, the next time through the loop).

*If you set a breakpoint and the program never encounters the breakpoint address (in other words, you push reset or the program goes astray), your code will remain corrupted by the JMP instruction and should be downloaded again.*

Do not set a breakpoint within 3 bytes ahead of any code that must be executed *before* the breakpoint is encountered. (This is common near the end of loops and subroutines.)

Use the **Loop** command only after encountering a breakpoint. The monitor must place another breakpoint at first opcode *after* the current breakpoint to execute the instruction *at* the break address and then go back to the original breakpoint (for the next time through the loop). Consequently: do not use a **LOOP** command from a breakpoint that is less than 5 bytes prior to a branch instruction.

## NOTES

In order to return to the EET monitor at the end of a program, use *LJMP 30H*. The program saves your most recent entry for each command. The EET monitor uses part of your target memory. During development, your programs should not use these areas. If a breakpoint is used, the monitor may destroy the data stored by your program.

### EET monitor-claimed internal memory

---

00H to 07H	Registers R0-R7
08H to 24H	User RAM (unless using register banks)
25H	<i>EET monitor Flags (bits 2EH and 2FH)</i>
26H to 3FH	User RAM
40H to 4FH	<i>EET monitor STACK</i>
50H to 7FH	<i>EET monitor storage RESERVED</i>

---

### EET monitor-claimed external memory

---

0000H to 1FFFFH	<i>Code Space: EET MONITOR CODE</i>
2000H to 3FFFFH	Code and DATA space for user programs
3F00H to 3FFFFH	<i>EET monitor interrupt vectors RESERVED</i>
4000 to FFFFFH	USER CODE/DATA space

---

Internal memory locations 00 to 07 make up registerbank 0,. Although the contents of R0 to R7 are saved and displayed with the other registers, the EET monitor cannot display these run-time contents with a *Display Internal 00, 07* command. You can use these locations but if you encounter a breakpoint, the run-time values of addresses 00 to 07 (as determined by your program) will appear in the Display Registers screen as R0 to R7, *not* in the Display Internal 00 to 07. When a program is running, however, the internal contents of these addresses will be the same as the contents of R0 to R7. This applies to the stack pointer and other SFRs as well. The register display screen separates the real-time register contents from the run-time register contents.

## EET Monitor Subroutines

The general-purpose subroutines of the monitor are available for the convenience of programmers who are doing development work. If you use these routines for development and then replace the EET monitor ROM with your code, it will lack the routines. You need to incorporate your own routines into your final code.

Name	Address	Function	Passed to	Destroyed	Passed back	Comments
SERIN	0C64H	Get one ASCII character from the terminal keyboard	nothing	A, R6	A = ASCII char	echoes to screen
SEROUT	0C50H	Display one ASCII character on the terminal screen	A=ASCII char	R6	nothing	
GHD	0C26H	Get two hex digits (1 byte) from terminal keyboard	nothing	A, R6, R7	A = 2 hex digits	forms a byte from two key-strokes
DISBYTE	0C3DH	Display one HEX byte (2 digits) on the terminal screen	A = byte to be displayed	R7	nothing	outputs two ASCII characters
PRINT	0083H	Print an ASCII message on the terminal device (outputs a table of ASCII characters)	DPTR = starting address of message	A, DPTR, R6	nothing	message must end starting with 00H
XDELAY	0D06H	Provides a time delay of 50 msec	nothing	nothing	nothing	
TIME	0D09H	Provides variable delays from 50 msec to 12.8 s in 50 msec units	B = number of 50 msec delays	B	nothing	Total delay time = contents of B $\times$ 50 msec

### Use of C Programs with the EET Monitor

You can download programs written in C using the EET monitor provided that they have produced Intel HEX files. The code *and* data will need to be located in the code/data space (usually at  $2000_{16}$ ). Code and data locations must not overlap (may not occupy the same addresses) since the addressing overlaps from  $2000_{16}$  to  $3FFF_{16}$ .

You can use interrupts by putting a jump to the interrupt service routine at the appropriate locations above  $3F00_{16}$ . Both the control of memory spaces and the interrupt jumps are usually managed by linking an assembly language module into the C program. The exact form of this module will depend on the C compiler used.

**Send31.** SEND31.EXE is a communications interface program you can use with the EET monitor program. SEND31 (as supplied) is expected to communicate with the EET monitor through COM1. SEND31 was written in QuickBASIC and may be modified as needed through QuickBASIC. The commands to start up the monitor from the PC are:

---

**SEND31 Enter** (invokes SEND31 to communicate with a development system)

---

**SEND31 fn.HEX Enter** (invokes SEND31 and automatically downloads the file fn.HEX to test)

---

**MON51 (Keil/Franklin).** MONITOR-51 operates on an 8051 microcomputer board. MONITOR-51 performs the communication with the PC system over the serial interface. Therefore, the operation of the monitor requires the Monitor-51 interface program MON51. The following commands are supported:

- Display the contents of the various 8051 memory spaces in hexadecimal and ASCII format
- Change memory contents interactively
- Display and change register and "Special Function Register" contents
- Initialize the various 8051 memory spaces with a constant value
- Disassemble the code area in 8051 mnemonics
- In-line Assembler
- Real-Time Go with up to ten fixed and one temporary breakpoint
- Control commands for up to ten fixed breakpoints
- Single-step execution, alternatively with the possibility to execute a subroutine as one step
- Download and upload for Intel HEX and Object files
- Command menu (help)

**Hardware and software requirements.** The following requirements must be satisfied in order to run the *MON51* program correctly:

- Hardware Configuration
  - IBM-PC/XT/AT or compatible systems
  - 128K-byte RAM
  - one floppy-disk drive
  - one serial interface (COM1)

- Software Requirements
  - MS-DOS or PC-DOS operating system, version 3.00 or higher
  - Assembler A51 for the generation of 8051 programs (optional)
  - C51 compiler for the generation of 8051 programs (optional)
- Target Requirements
  - 8051 CPU (for example, SIEMENS 80535)
  - 5K-byte EPROM starting at address 0 (loaded with MONITOR-51 software)
  - minimum of 256 byte RAM (von Neumann wired, this means that access is possible from XDATA and CODE space)
  - serial interface for terminal communication
- Other Requirements
  - serial interface
  - additional 6 bytes stack space in the user program to be tested
  - 256 bytes external data memory (RAM); additional 5K bytes with use of the trace buffer
  - 5K bytes external code memory (EPROM)

All other hardware components can be used by the application.

**Starting MONITOR-51.** *MON51* is invoked by entering its name: '*MON51*'. The communication with the serial interface is performed with the hardware interrupt. Various parameters can be stated in the invocation line. These parameters change the working method of *MON51*.

The general syntax of the *MON51* invocation is as follows:

---

*MON51* [parameter]

---

where:

---

<b><i>MON51</i></b>	is the program name
<b><i>parameter</i></b>	is one or more of the following:
<b><i>COM1:</i></b>	use serial interface COM1: (Default). Abbreviation: 1.
<b><i>COM2:</i></b>	use serial interface COM2: Abbreviation: 2.
<b><i>COM3:</i></b>	use serial interface COM3: Abbreviation: 3.
<b><i>COM4:</i></b>	use serial interface COM4: Abbreviation: 4.
<b><i>INT14</i></b>	the serial interface is called through the BIOS software interrupt 14H. No hardware interrupt is performed. The abbreviation of <i>INT14</i> is 1.
<b><i>NOINT</i></b>	the serial interface is polled. No hardware interrupt system is necessary. The abbreviation of <i>NOINT</i> is N.
<b><i>BAUDRATE ( bps )</i></b>	allows the setting of the transfer baud rate. If this option is omitted, the baud rate 9600 bps is used. The value (bps) specifies the baud rate. Possible values are: 300, 600, 1200, 2400, 4800, 9600 and 19200. The abbreviation of <i>BAUDRATE</i> is BR.

---

All invocation parameters of *MON51* can also be executed through the system variable "*MON51* =", which can be installed with the help of the DOS command SET. If no parameters are stated in the invocation line, *MON51* automatically operates with the parameters of this system variable. *MON51* outputs the following message if the parameters of the system variable "*MON51* =" are used:

After invocation, *MON51* signs-on with the message:

---

```
MS-DOS MON51 Vx.y
      INSTALLED FOR PC/XT/AT (COMLine 1) USING
      HARDWARE INTERRUPT SERVICE
BAUDRATE 9600 (DEFAULT)
*** MONITOR MODE *** or ***TERMINAL MODE***
Vx.y is the actual version number of the MONITOR-51
      software.
```

---

After *MON51* has been invoked, test is made whether MONITOR-51 is active on the microcomputer board. If this is the case, *MON51* switches to the "Monitor Mode" and commands can be entered. Otherwise, an assumption must be made that a program is already running on the board. In this case, a switch is made to the "Terminal Mode" in order to input or output data via a serial interface.

**Editing a command line.** Command lines can be manipulated using the following control keys:

---

<i>Enter</i>	Executes entire entry line
<i>Backspace</i>	Deletes character in front of the cursor
<i>DEL Ctrl+D</i>	Deletes the character below the cursor
<i>Ctrl+A</i>	Deletes all characters to the right of the cursor
<i>Ctrl+X</i>	Deletes all characters to the left of the cursor
<i>Ctrl+Z</i>	Deletes entire entry line
<i>Escape, Ctrl+C</i>	Aborts entry and starts new entry line
<i>Home</i>	Positions cursor at the beginning of the entry line
<i>End</i>	Positions cursor at the end of the entry line
<i>Insert</i>	Toggles between insert/overwrite mode
←	Move cursor one position to the left
→	Move cursor one position to the right
↑	Assumes a prior entry line in the editor
↓	Assumes a later entry line in the editor

---

**Controlling console output.** The following control the output on the console:

<b>CTRL-S</b>	Stops output to the console and suspends the execution of the current command
<b>CTRL-Q</b>	Cancels the effect of CTRL-S. Output to the console resumes.
<b>CTRL-C, ESC</b>	Terminates the execution of a command and returns to the command line interpreter. MONITOR-51 produces the following console message *** TERMINATED *** #

## MON51 commands

Command	Description	Example(s) You type <i>italic</i> entries
F1	Terminate MON51 and return	<F1> <b>EXIT MON51</b> (y or [n]) y <cr>
F2	Transmit contents of a file (the file contents is interpreted as commands to the monitor)	<F2> <b>Input File:</b> MYINIT.CMD<cr>
F3	Protocol the screen output (all characters showing on screen also go to designated file)	<F3> <b>Output File:</b> DEBUG.PRN<cr>
HELP	See brief description of all commands	<b>#HELP</b> <cr>
:	Comments (useful when in Protocol (F3) mode)	<b>#;</b> <i>this is a comment</i>
EXIT	Leave monitor and return	<b>#EXIT</b> <cr>
D	Display Memory (no address—displays same as last use of command; no second address—display from first address to end of memory space)	<b>#DB</b> <b>#DC</b> 0 ,26 <b>#DD</b> 30 <b>#DI</b> 70,90 <b>#DX</b> 200,0BFF
E	Change memory interactively (return—the contents are unchanged and pointer jumps to next address; period (.)—end of command sequence;)	<b>#ED</b> 27 18 <cr> <b>28:</b> 3F 0B <cr> <b>29:</b> 4A . (no change to data 27 or 29 but 28 changes to 0B)
F	Fill memory—blanket fill of a space with the same value	<b>#FILLC</b> 0,3FFF,23 <b>#FILLD</b> 0, 7F, 0 <b>#FILLX</b> 400, FFF, 7F
U	Unassemble—displays code in assembler mnemonics	<b>#U</b> <b>#U</b> 100 <b>#U</b> 2000, 20FF
A	Assemble—insert in-line assembled code in place of existing instructions	<b>#A</b> 8000 <cr> <b>8000 SJMP 8010</b> MOV DPTR,#4000 <cr> <b>8003 NOP</b> MOV RO,PCON <cr> <b>8005 NOP</b> MOV A,R1 <cr> <b>8006 CLR A</b> <cr> ;take over instr



Command	Description	Example(s) You type <i>italic</i> entries
		<b>8007 CLR A</b> <i>MOVX A, @DPTF</i> <i>&lt;cr&gt;</i> <b>800B ANL P0, #12</b> <i>MOV</i> <i>@R1, A &lt;cr&gt;</i> <b>8009 RET</b> <i>INC DPTR &lt;cr.&gt;</i> <b>800A INC R1</b> <i>MUL AB &lt;cr&gt;</i> <b>800B NOP</b> <i>. &lt;cr&gt; ;termi</i> <i>nate</i>
X	eXamine: change and display registers	<b>#X &lt;cr&gt;</b> <b>RA RB R0 R1 R2 R3 R4 R5 R6</b> <b>R7 PSW DPTR SP PC</b>
B	Breakpoint commands	<b>#BS 8100&lt;cr&gt;</b> <i>(set break-</i> <i>point)</i> <b>#BK ALL&lt;cr&gt;</b> <i>(clear all</i> <i>breakpoints)</i> <b>#BK 2&lt;cr&gt;</b> <i>(clears 2nd</i> <i>breakpoint)</i> <b>#BL&lt;cr&gt;</b> <i>(list breakpoints)</i> <b>#BD 1,3,5&lt;cr&gt;</b> <i>(disable,</i> <i>but keep, breakpts)</i> <b>#BE ALL&lt;cr&gt;</b> <i>(re-enable</i> <i>breakpoints)</i>
G	Start program execution	<b>#G&lt;cr&gt;</b> <b>#G 2000&lt;cr&gt;</b> <b>#G 2000, 20FE&lt;cr&gt;</b> <i>(break</i> <i>at 20FE)</i>
T	Trace execution--single-step mode	<b>#T&lt;cr&gt;</b> <b>#T25&lt;cr&gt;</b> <i>(take 25 steps--</i> <i>each step displays)</i>
P	Procedure trace--subroutines execute as single step	<b>#P&lt;cr&gt;</b> <b>#P5&lt;cr&gt;</b>
LOAD	Download a program in HEX or binary format	<b>#LOAD TESTPROG&lt;cr&gt;</b>
SAVE	Save program in Intel HEX format	<b>#SAVE MYPROG.HEX 2000,</b> <b>20FE&lt;cr&gt;</b>
LS	Load symbols of special function registers	<b>#LS REG517. INC&lt;cr&gt;</b>

Again, I omit many details. One feature of particular interest with the MCB520 board is the ability to download code located at 0000. Some features in the address decoding allow the monitor, which resides at high address, to be reached on power-up, yet allows the monitor to start the user's code.

## Emulators

In-circuit emulators are tools that allow you to develop code with debugging capabilities while running it on the actual target hardware. Unlike a simulator that requires you to generate any I/O signals in software, the emu-

lator can use the actual target hardware. An emulator, like the one sketched at the start of this chapter, has a cable that goes from the socket that will hold the microcontroller to (usually) a PC add-in card. The resources of the PC are available to edit and compile programs and then, depending on the speeds involved, the code is either run directly from the PC or else loaded into memory on the emulator card. The emulator tool sends and receives information from the socket pins as though a micro were there.<sup>37</sup> The real challenge, not a problem with the 8051 speeds, is to emulate at real time—when the processor is running as fast as technology allows, the additional circuitry must be faster to monitor the activity. I omit details here because most emulators are quite expensive for home or class use.

## PROM Programmers

Somewhere in the development process, you will have to burn in programs. Some of the development boards will burn the EPROM within the micro or even the separate, generic EPROMs right on the board. For others, you need a separate programmer. You can check some of the magazines for such devices. Some time ago, I used a low-cost one from Needham Electronics.<sup>38</sup> I suppose that significant software changes have come about with Windows-based interfaces, so I will not go into much detail. A product from another company may have a significantly different user interface.

Having installed the card and shelled out to DOS, enter the software by typing:

---

```
C : >PBIO\EMP
```

---

This brings up the menu screens. With the Needham unit, this involves a two-level menu. You specify such things as the particular type of device to program, the file location for the source code, and the format for the data. For example, the device select menu (click the mouse on it) picks the manufacturer (perhaps Intel or Philips), and the next level picks the particular de-

---

<sup>37</sup>The emulator may either simulate the processor with other hardware or else have additional hardware to “watch” the activity to an actual microcontroller on the card. The challenge is to see what is going on *inside* the micro, which is why the less expensive emulators only work with devices having external code. The more expensive ones use “bond-out” chips, which bring additional signals out that would normally not be available around the edges of the microcontroller.

<sup>38</sup>See Appendix A6 for addresses and telephone numbers of development hardware suppliers.

vice.<sup>39</sup> You also enter the file name, *A:myfile.hex*, and the file type (Intel's .HEX).

It is essential that you understand the *buffer*. You load your HEX file from your disk into an area of PC memory (the buffer). Then you can reassure yourself that the data looks about right (or even manipulate the contents of individual bytes) before programming. You may want to clear (to FF) the buffer *before* loading code so the programmer does not try to program high-address code left over from some earlier use of the buffer.<sup>40</sup>

Once the content of the buffer is reasonable, you request the programming of the EPROM. With the Ncedham unit, you then see a picture of the cable configuration and adapters that may be required as well as an indication of which way the device goes in the socket. Be careful to add or remove adapters in the proper orientation and install the chip the right way at the right end of the socket (do not rely on intuition since some chips go in "backwards"). When that is set, either verify that the EPROM is erased, or let the programming proceed. The programming process is quite quick for 2k to 8k programs. If anything does not "take," the process stops with an error message. Usually this indicates that the device was not fully erased at the start. If the device is empty but refuses to program or if it refuses to erase in 15 to 20 minutes in a UV eraser,<sup>41</sup> you probably have a bad device. That seems to be a not-uncommon occurrence with certain types of devices, but I have never established whether this is due to poor handling, incorrect erasing, or defective devices. I believe some manufacturers suggest EPROMs should function for 100 cycles.

## DEBUGGING STRATEGIES

Development tools are only useful if you have a strategy for debugging. In the pages that follow are hints that I have found to be helpful, both person-

---

<sup>39</sup>Note that a 2764 is *not* at all the same as a 27C64 because it programs at a much higher voltage.

<sup>40</sup>You ought to examine (edit) the buffer to be sure the right sort of code is starting at 0000h (be sure you aren't burning in a file starting at 2000h because you still have the linking set for downloading to a high address). It is particularly useful to check for the long jump (02) to the working code at address 0000. If you are using the 750/1 you want to be sure that there is *not* an 02 because that is a *long* jump (not supported). If so, you still haven't gotten the ROM(small) control to take effect.

<sup>41</sup>A UV (ultraviolet) light for erasing EPROMs through the clear window is a necessity if you are to reuse them. Special units with a timer are not too expensive and are available from many distributors including Jameco and Digikey.

ally and for many of my students. The sad part is that these strategies always have to be learned by experience—I try not to think of the foolish mistakes I have made over the years.

## Test Hardware FIRST

**Test the hardware before you test the software.** With embedded controllers, software depends on understanding the hardware. All the clever software in the world will fail if the ports have failed or the polarity of a *start-convert* or *enable* pulse has been crossed up. The functioning of timer and interrupt hardware can be vital to anything else working.

## Test the Processor

First, you need a way to tell if the micro is alive.<sup>42</sup> With a home-built single-chip or several-chip system, start by looking for the *ALE* signal with either an oscilloscope or a logic probe. If the signal is present, you know the crystal oscillator is running and normal program fetches are occurring. For single-chip applications one of the most common errors is to not ground the *EA* pin. Other common problems relate to burning the code at the wrong address (still at a high address for the monitor) or without the special requirements for *ROM(small)* with the 750/1.

## Test the Ports

Particularly if you are debugging with a monitor, I suggest you *keep an arsenal of little programs* you can download to make sure that “*one plus one still equals two*.” Here are short test programs I would write and keep on hand:

1. **Drive all the output ports:** I would begin by driving all the outputs high and then low at a 100 kHz rate so you can easily attach a scope (or a logic probe) and verify that the outputs are toggling. To get fancier and test wiring order, I would have a test program that drives pins high in succession from bit 0 through bit 7. Better yet, I would drive bit 0 high, then bits 0 and 1 and then bits 0, 1, and 2 and so on. It is very easy to tell which bits are which that way.

---

<sup>42</sup>If you are using a development board with a monitor, it is easy to tell if all is OK because the monitor will only respond on the PC screen if both the microcontroller and the serial communication links are healthy. (If not, it is often in the configuration of the COM ports on the PC.)

2. **Test the A-D:** I would read in from each analog input in succession and output to a *corresponding* bit on a parallel port a 1 or 0 depending on whether the reading is above or below midrange. Then I could attach a potentiometer and a logic probe and get an indication of basic functionality of the A-D. Even simpler, rather than a potentiometer, I might just tie the input to 0V and then +5V and look for the corresponding port bit to change. If it is important to test the A-D in detail, you could simply output the value of the reading from a specific analog input pin to an entire parallel port (8 bits). You could watch (perhaps with LEDs) the reading change as you change an input voltage value with a potentiometer. In my experience, if a port or converter is bad, it is usually *very* bad, so a basic functioning test is sufficient.<sup>43</sup>
3. **Test the timers:** I would start with a simple time delay that I used to toggle a port pin. Internal timers do not usually fail independently, so if the micro is alive, the timer is usually OK. The problem is more whether you have set up the timer correctly, hence the value of testing that piece of software.
4. **Be sure interrupts work:** This might be something as simple as pushing a button to cause a hardware interrupt and having a port bit change within the interrupt routine. If program flow gets there, you see the result—if not, the bit stays the same. Again, if that works, probably all the interrupt functions are intact.

## Growing Software

You may wish to start with the *simulator*. If you have program-intensive (as opposed to I/O-intensive) programs or routines, it is wise to start there. If much of your program involves complex or unpredictable I/O signals, you may go directly to the hardware. If you have a PU552 or MCB520 board, your best approach may be to start with downloading. That allows you to examine breakpoints and the values in variables as you go while still interacting with actual external hardware.

## Start Small

However you develop software, do not wait until it is “all written” before trying it on the target microcontroller. Tempting as it may be to stay in the theoretical writing phase, you can be much more confident if you start

---

<sup>43</sup>It is not uncommon to have single bits of a port fail while the rest of the port continues to function. But when a bit has failed there is no question—it is never 25% bad.

with small skeletal pieces of the overall program that work with the hardware. Then you can go back to the grand plan of the software with confidence. You can grow the program in a systematic manner rather than entering it all and then wondering which problem to fix first.<sup>44</sup>

### **Avoid “Burn and Try” (Develop First on a Higher-Powered Relative)**

If you are aiming to *end up* with a single-chip wire-it-yourself solution, I strongly suggest you avoid the “burn and try” approach. Some chips will not handle more than a few tens of erasures and erasing takes time.<sup>45</sup> In addition, without an emulator or the tricks of the next sections, a burned program has no debugging help. You cannot tell what is going on inside if things do not somehow show on the I/O. Instead, develop with a monitor program on a bigger board. Specifically, you can purchase a PU552 or MCB520 board and cable from the ports on the development board to the pins on the socket where the *final, single-chip* controller will go. Then you can wire all the rest of the hardware out from the socket and still be able to download and test with a monitor.

### **Use Breakpoints**

One way to debug software when it is in RAM, running under a monitor program,<sup>46</sup> is to use breakpoints. In essence you insert traps at one (or more) carefully chosen places and start your program running. When (or if) the program flow reaches the breakpoint, control reverts to the monitor. You immediately know that the program has neither stuck in an endless loop nor branched some other way and missed the breakpoint. You can then examine the contents of memory to see if they reflect the conditions you anticipated. If flow does not reach the breakpoint you can only stop the flow with a reset button, but you do know flow did not reach that place in the program.<sup>47</sup>

With a good choice of breakpoints you can see the outputs as they are issued or the inputs as they come in. You can check calculation algorithms

---

<sup>44</sup>I know all this because I, too, have spent lots of time doing development the wrong way. We used to speak jokingly of the engineer with “twenty years of experience—all of it had.”

<sup>45</sup>About fifteen to twenty minutes at the recommended UV intensity.

<sup>46</sup>Breakpoints are common to simulators and emulators as well.

<sup>47</sup>Incidentally, since a monitor has to modify the code in your program to insert a break (which it restores when the break is reached), if you *reset* instead, you should download your program again—the break code will still be there, but the monitor will not have the restoration data.

to be sure that they are doing what you intend. Once you have written the best software you can, the use of breakpoints is about the only way to discover what you overlooked.

### Use I/O Pins as Scope Trigger Points

If you have unused I/O pins, you can do simple debugging with temporary code using the pins to report what is happening. For example, you could set a pin high each time you enter a certain function and clear it when you leave. Just looking at the pin with a scope or logic probe can tell whether you enter the routine and how much of the time the program spends there. Whatever you need to know about program activity, with a few pins you can temporarily monitor it non-intrusively through extra port pins, while it continues to run (as opposed to breakpoints where flow stops as soon as you reach one).

The interrupt is one of the more difficult features of embedded software to test.<sup>48</sup> With all the register setup requirements, it is common to overlook something and never reach the interrupt.<sup>49</sup> The best way to test interrupts is to write very simple ones first. You may do significant calculations later, but for now just have code to toggle a port bit or alter an extra temporary variable in the interrupt—something you can easily check with a breakpoint or with a scope or logic probe. Once you have the assurance that the interrupt is really occurring, you can go on to other things. Until then, you can never be sure whether a problem is in the interrupt or due to never reaching the interrupt.

---

#### Simple program to test Interrupt functionality with Port pins

```
#include <reg51.h>
char i;
```

---

<sup>48</sup>Again, a simulator is messy to use here since, by its very nature, an interrupt comes in from some outside stimulus at no particular relation to the program flow. Emulators can be better, but it is again a function of how they are handling the code and how they are set up.

<sup>49</sup>This is particularly likely with a monitor where the interrupt code must be located at a specific place in RAM pointed to by the actual interrupt vectors in the low-address ROM area shared with the monitor code. In some monitors, your actual interrupt code must be reached by “a hop, a skip, and a jump”—the actual vectored call to a fixed-address pointer in the bottom forty or so bytes of unalterable monitor ROM space (int0 at 8, int1 at 16, and so on), the jump to a fixed address in modifiable program (dual-addressed RAM) space, and finally the jump to the actual interrupt.

```
void msecdel(msec)int msec;{
    int x,y;
    for (x=1;x=msec;x++){
        for (y=0;y=10000;y++){
        }
    }
}

void main(void){
    for(;;){
        for(i=1;i<=10;i++){
            P1=i|0x10;
            msecdel(10);
            P1=i;
            msecdel(10);
        }
    }
}
```

---

If you have several interrupts, you can issue a code out the port pins to identify the particular interrupt running now. If you have an interrupt-driven system such as the scheduler in Chapter 11 it is important to know how much free time the processor has. If there is no background task that must be running (the main program), you can get a measure of the amount of time *not* spent in interrupts by putting a counter in the background instead of an idle loop. Then, by toggling a port bit every time the counter folds over, say, you can get an indication of how much time the processor spends outside of the interrupts.

## REVIEW AND BEYOND

1. What is the purpose for a *development environment*? Can you compile programs without one?
2. How does a *make* file simplify program development?
3. What is the one menu command to update all the files and get a finished file to try out?
4. Explain how colors can help you locate a missing *\*/* in a comment line.
5. Why are you advised to test hardware before testing software?
6. Explain the differences between developing with a simulator, an emulator, and a monitor.





## SECTION III

# MULTITASKING

---

Perhaps you think multitasking is an “advanced” method. In this section, I hope to change your perspective as I show you this fundamental part of efficient programming deserves your attention. With multitasking you can have your microcontroller seem to do many things at the same time—the microcontroller will never seem unavailable and will never be sitting idle when there is something to do. Once you understand the concepts of multitasking, you will be able to better appreciate what goes on with Windows® programming on a PC.

Chapter 10 introduces the terms and describes the various classifications for multitasking (or “real-time”) operating systems.

Chapter 11 picks up with the key internal hardware—the timers and interrupts. While they can be used for many other things, in my opinion their combination into a real-time clock is the most important development since programming started.

Chapter 12 gets specific by developing a scheduler—one of the simplest real-time systems—and applying it to a traffic light and then to a solenoid cyler such as might control an envelope-handling system.

Chapter 13 introduces real-time operating systems (RTOS) that standardize the way one writes tasks and handles their interactions. This is covered in much more detail in the companion book.



# 10

## Concepts and Terms

---

### BEYOND SINGLE-PROGRAM THINKING

Up to this point, all the C and assembly language examples have used traditional techniques of microprocessor programming. When your applications grow more involved and time-critical though, the single-program approach becomes awkward to handle. If you have several external hardware devices that you need to control at the same time, the challenge is to make sure you satisfy each device.

Programming was different when only one thing happened at a time. If an LCD alphanumeric display module needed to be cleared, for example, you put out the appropriate code, pulsed the enable or write line, and waited long enough for the clearing process to finish,<sup>1</sup> and then went on to send a new character. You could not do anything else while the program was waiting.

To progress to multitasking you must develop a *new way of thinking* about program flow that does not let it become stuck if something is not yet ready. With multitasking, you can spend the time the display is clearing scanning the keypad, starting an A-D reading, or computing the average of a string of readings. If nothing else, you can be *alert* for other inputs that may arrive unexpectedly.

---

<sup>1</sup>Or you might have checked the status code if you were using the read-back features.

## WHAT IS “REAL-TIME”?

In this book *real-time* refers to any system that responds to inputs and supplies outputs *fast enough* to meet hardware or user requirements. This section describes systems that illustrate the range encompassed by “real-time.”

- 100 msec:** A keyboard entry system is real-time if it gives some response to users quickly enough so they feel confident that the system “noticed.” Just a *beep* within 100 msec makes a user confident that the system recognized the input. In this time frame, because humans can’t push new keys any faster than 10/s,<sup>2</sup> a key scanning routine that repeated every 100 msec would probably meet the requirement of *fast enough*.
- 200 msec:** The human visual processing system cannot assimilate new digital display information more quickly than perhaps five times a second, so updating a display of rapidly changing numeric values only a few times a second is *fast enough*.
- 1 msec:** Machines are more time-critical than people. A moving stepper motor needs new step pulses fairly consistently—here millisecond precision is *fast enough*.
- 1 min:** Serial ports usually have only a one-character buffer, so, at 9600 baud, if every *incoming* character is picked up within about 1 msec,<sup>3</sup> that is *fast enough*. *Outgoing* (asynchronous) characters can go whenever the UART is not busy so, depending on what application is waiting for the information, sending characters within a minute might be *fast enough*.
- 100 µsec:** The same considerations would apply to collecting data with an AD converter. Depending on how rapidly the incoming voltage is changing, retrieve the reading immediately or let it sit until you have time to process it.

---

<sup>2</sup>Technically a user could push several keys at once more quickly. If the routine to scan the keypad can recognize multiple key presses as well as individual releases, it can catch all the new information. If multiple keys are a problem, the routine can just ignore *all* keys when the user pushes several at once and make the lack of response a means to train the user to not expect results if he or she jams down a bunch of keys at once! No one said you have to accept garbage in! The exception would be a *keyboard* where typists are used to going fast enough to sometimes have several keys down at once in succession—called *n-key rollover*.

<sup>3</sup>10 bits @9600 bits/sec = 960 char/s or about 1msec/char.

*How rapidly should you adjust a flow valve for a process? How quickly should you supply a new speed setting to a motor?* These questions relate to what is *fast enough*.

At the other end of the spectrum from real-time systems are the traditional ones that do data processing for large organizations—updating insurance records or inventory lists for example. They take a batch of input data from a file and store the results in another file or send them to a printer some time later. Whatever the data and whatever the processing, there is a significant delay between the request for processing and obtaining the results. If the delay were less, presumably it would be more useful or make the user more effective. If a data processing job takes 20 s, it might be better if it finished in 1 s. For a job normally finished 100 msec from when the operator pushed the last key, it is doubtful that speeding it up to 1 msec could even be perceptible to a human. Overnight processing in this case is not *real-time*, 20 s processing is not *real-time*, but 1 s or less is probably *real-time* to a human user.

Some areas where there is a push for real-time systems include speech recognition, image processing, and digital signal processing. Imagine a dynamic display of the spectral energy distribution of a speech signal as a microphone picks it up rather than later from processing a recording! Consider the possibilities of issuing canceling signals to make an airplane invisible to radar or an acoustic system to cancel background noise. How about recognition systems that can identify objects as they travel down a conveyor line to direct a robotic system?<sup>4</sup>

## JOB

A *job* (or perhaps call it a *project*) is the overall thing to do. For a dedicated microcontroller, the job is the total mission for the device and its associated hardware.<sup>5</sup> A job might be to control a microwave oven. It consists of multiple *tasks* (hence the term, *multitasking*) such as scanning the user touch pad, updating the displays, tracking the time, monitoring the safety interlock devices, and driving the Magnatron.

---

<sup>4</sup>Much of the high-tech focus of the microcontroller field involves very compute-intensive applications. In this book, I only discuss much less demanding parts of the real-time application world. Some of the high-tech applications are impossible in real-time with a single 8051, but the difference between the large, fast applications and an 8051 job is more one of scale than of technique.

<sup>5</sup>As you should know from the previous section, the term *project* is used by the  $\mu$ Vision environment to describe the overall combination of the individual *software* modules. To distinguish this software-only perspective from the overall hardware-software combination, I will use the term *job* for the latter.

## TASK

A *task* is *one* thing to done by a controller. Tasks represent a *way of thinking* that logically divides a job and leads to the effective use of the microcontroller. Imagine a human analogy: the overall mission or job could be, *Drive out the dictator*. The tasks might be, *Drive this supply truck to the front lines*, and, *Sit in this trench until ordered to advance*. For a microcontroller, tasks might be, *Wait for and process input from a keyboard*, or, *Keep a motor running at a set speed*. A task can be a single function having a few lines of code, or it can be an entire set of interrelated functions. The task divisions are arbitrary but they ought to relate to functional parts of the overall mission.

### Priority

A *high-priority* task is more important or urgent—it should go first. Having different priorities in a system implies that some activities are more urgent or important than others. The idea is common to politics and government—we expect to yield to fire trucks and ambulances because they have urgent missions. The same thing applies to microcontroller tasks.

Assume there are several tasks that need processor time. Should a given task “move to the head of the line” if other tasks have been waiting longer to run. Sending a pulse to a stepper motor probably is more urgent than computing an average—stepper pulses should come at regular times, whereas you can do computation whenever time is available. In multistage processing of incoming data, if there is a chance incoming readings can overwrite older readings, then first-level processing to make room for incoming readings has higher priority than later stages of data compression. Priority can reflect importance or urgency, or it can reflect duration—give a very short task higher priority in the same way that a kindly person at the grocery checkout with a cart full of groceries might allow a person with only a jug of milk to go ahead.

## PREEMPTION

Sometimes a task should not just move to the head of the line, but should even “kick out” the task that is currently running. If so, it has *preempted* the running task. The concept is not difficult but the implementation in software can be challenging. If the task you are going to preempt is in the middle of something, either do not interrupt it or, at a minimum, save the registers so it can resume from where it halted.

For a human analogy, imagine what would happen if the checkout person had already started handling the cart full of groceries when the person arrived with just the jug of milk. To preempt the current batch would require saving the current total, moving groceries aside, handling the jug of milk, and then somehow resuming the previous checkout. Obviously, the grocery checkout is not preemptive!

## MULTITASKING

Getting the computer to manage several tasks seemingly simultaneously is the heart of real-time *multitasking*. The ideas involved are not difficult to visualize, but they differ from those of normal programming. Develop the frame of mind that says, “*If it isn’t ready yet, proceed to the next task. I will come back to it later. Right now there may be something else to do.*” With multitasking you can do data processing and other computing in the background (in the main program, for example) while you do *time*-related things—refreshing a multiplexed display, scanning a keypad, watching for the arrival of a slow event—in interrupt routines or scanning routines driven by a regular interrupt.

There are several fundamentally different ways to do multitasking. I develop a *scheduler* in Chapter 12. In this book, I only briefly describe several commercial multitasking operating systems.<sup>6</sup>

### Cooperative Multitasking—Round Robin

The simplest form of multitasking is *cooperative multitasking* where each task gets its turn. Call this approach “multiple tasking.” It is a *round-robin* system where tasks run one after the other in turn, without the idea of preemption or priority.

Although this approach may include interrupt routines, program flow focuses on the *background* tasks. Each task runs in turn. If a task has nothing to do when its turn comes, it passes control to the next task. If it has a lot to do, you allow it to spend all the time it needs to finish. You must write any long tasks so they pause at frequent points in their program flow, allowing other tasks to get a chance to do some work. The weakness of the scheme is that, if you inadvertently add some task that does *not* cooperate,

---

<sup>6</sup>In the companion book I develop several more elaborate approaches from the code level.



there is no protection in the operating system, and you have blocked all the other tasks.

This approach avoids the overhead of an operating system, but requires close attention to avoid long *latency* (delay in getting back around the loop to the task needing attention). To make it work well, you must break big tasks into several small tasks or program in intermediate pauses to yield to other waiting tasks.

### Time-slice Multitasking

The *time-slice* approach to multitasking solves some of the problems of cooperative multitasking. Time slicing provides protection from hogging the processor by arbitrarily switching out long-running tasks at regular intervals. With this approach, tasks run round-robin unless a task takes more than an allotted amount of time. At that point, with no action on the task's part, the operating system sets it aside and runs the next task in line. The first task can resume running when its turn comes around again.<sup>7</sup>

For this approach, you need a timer-driven interrupt to mark off time slices to determine when to remove control from the running task. In its basic form though, time slicing does not allow *preemption* within a time slice—needed for *nondeterministic* systems where urgent tasks may unexpectedly or unpredictably need immediate attention.

### Multitasking with a Scheduler

A *scheduler* resembles a time-slice system in that it keeps track of time for the various tasks. It works best where there are short, regularly occurring tasks, such as scanning external inputs or refreshing outputs. These tasks may come frequently or only occasionally, but they need to run promptly. A scheduler allows these tasks to execute at regular intervals—perhaps at every interval, every third one, or every hundredth one.<sup>8</sup> Any leftover processor time you can use for a background task that involves less urgent data processing or decision-making activity.

---

<sup>7</sup>This approach is good for data processing applications and was the heart of *time-shared* computing systems that were common before the personal computer revolution.

<sup>8</sup>This interval can be created by a timer and interrupt—the *real-time interrupt* discussed in the next chapter.

## Priority-based, Preemptive Multitasking

There are a confusing number of combinations when you begin to add in *priority* of tasks. More urgent tasks can go to the head of the line or preempt the running task. A common combination is a *priority-based, preemptive multitasking* operating system where *equal* priority tasks run in round-robin fashion while higher-priority tasks preempt ones of lower priority.

## Events

An *event* is anything that might cause the system to change between tasks. It could be periodic like the tick of a clock, or asynchronous based on input from outside hardware. An event could be an *external* interrupt, an *internal* signal from another task, or the expiration of a waiting period. Any system where the processor activity varies depending on external signals is an *event-driven* system.<sup>9</sup>

## REAL-TIME HARDWARE REQUIREMENTS

All members of the 8051-microcontroller family contain the two hardware features that are essential for real-time systems—timers and interrupts.<sup>10</sup>

### Timer

A *timer* can generate regular intervals that allow things to happen on time. A timer is actually a counter running off the microcontroller's clock. Since it keeps on counting independent of the software that the processor might be running, you can manage time delays without being dependent on software delay loops.

Whether or not they involve priority or preemption, the multitasking systems in the last three categories (time-slice, scheduler, and preemptive) all keep track of time for the tasks they manage.

---

<sup>9</sup>While technically a data processing system that does the payroll one time and inventory another time depending on a few operator inputs and the loading of tapes could be said to do different things based on user inputs, it is not considered *event-driven* because, once it starts a job, no unexpected user inputs are involved.

<sup>10</sup>Details of the hardware and its initialization are found in Chapter 11.

## Interrupt

Along with a timer, a real-time system should have at least one *interrupt*. You may think of interrupts as caused by signals from external hardware, but the most useful interrupt is one caused at regular intervals by the timer overflowing. Such an interrupt can break the processor out of the current task so it can see if anything *else* needs attention. With such an interrupt you do not have to do anything special in the individual tasks to be sure each gets a share of the processor's time.

## Real-time Clock

A *real-time clock* is the combination of a timer and an interrupt. Suppose the timer regularly causes an interrupt every millisecond. If you keep a count of the number of interrupts, where a stepper motor should step every 10 msec, you can issue the step pulse every tenth interrupt. If you need to scan a keypad every 40 msec, do it every fortieth interrupt. That way you avoid repeated polling or wasting hardware interrupts for slow events. Chapter 11 shows the programming for a real-time interrupt with an 8051.

## PROGRAMMING HABITS FOR ALL MULTITASKING

When you go on to write tasks for the scheduler developed in Chapter 12,<sup>11</sup> keep the following programming concepts in mind. These are the outworking of the *different way of thinking* promised earlier.

### Never Wait

With the possible exception of *preemptive* operating systems, always write program pieces as though you either do an activity at once or skip it until a later time. Like catching a bus, either you are there when it arrives, or it goes without you and you must wait for the next one. Imagine how it would be if the bus waited whenever someone might want to catch it *in a few minutes*. In the same way, if a task *waits* for a user to push a button or creates a delay in a counting loop, everything else will have to wait—this is called *hogging the processor*.

---

<sup>11</sup>Also, the concepts apply to more elaborate operating systems such as the commercial ones introduced in Chapter 13 or developed in the companion book.

## Set Flags

Whenever you need to remember that something has happened, but cannot respond right away, set a flag or increment a counter. Often the event needs to start some longer, more processor-consuming activity, and now may not be the best time. You may need to scan other inputs before going on to finish the processing for the first event. When you finish the urgent activities, have a background task check all the flags to see if there is any time-consuming processing remaining. The quick setting of flags records the events without delay, and then the background task does the lengthy processing as time becomes available.

## Be Sure Variables Are Global

With C, the compiler may overlay local variables (used only within a given function) and there is no guarantee they will still have their last value the next time you call the function. Either make the variable *static* or else make it *global* to the module.

## Avoid Software Loops

Use the timing features of the microcontroller rather than the software-generated delays produced by delay loops. Even better than using the timers directly, use one timer/interrupt to manage all the timing for all the tasks. You only need *one* timer, not a timer for each delay function in use at a given time. As long as 1 msec is an acceptable time increment, you can manage all time-critical activities with the real-time clock—at 1 msec if you do something every 1000<sup>th</sup> interrupt that is once a second or every 10<sup>th</sup> interrupt that is 100 times a second.<sup>12</sup>

## REVIEW AND BEYOND

1. Define “real-time.” Describe two or three situations where “fast enough” is not very fast.
2. Define “multitasking,” and compare it to parallel processing and conventional programming.

---

<sup>12</sup>Sometimes for very short delays it makes sense to have a software delay loop—perhaps if you need a 10 to 20  $\mu$ sec delay, for example, when your real-time clock is ticking at a 1 msec interval.

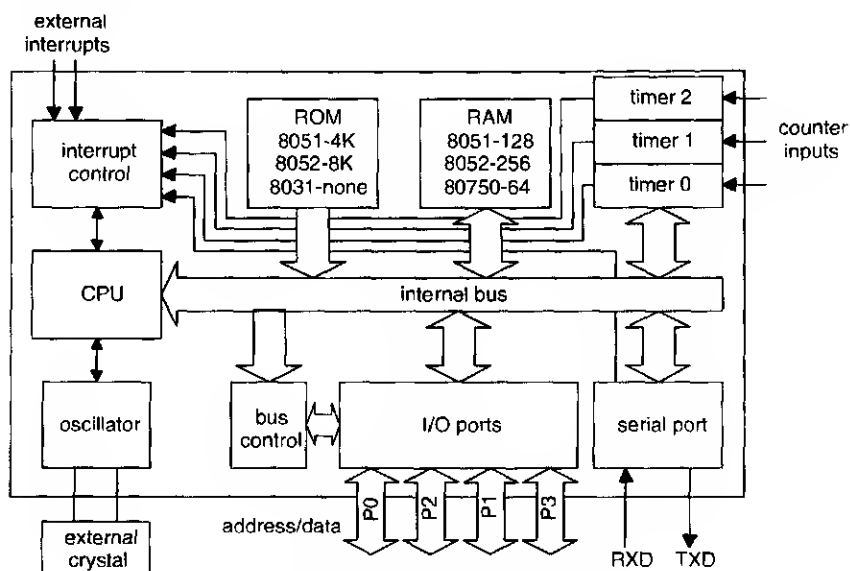
3. What are the advantages and disadvantages of multitasking over single-program solutions?
4. Why is the idea of a task helpful in project planning?
5. What is context switching? What feature in the 8051 was intended to speed context switching? How is it normally done in other computer systems with “flat” memory architecture?

# 11

## Timers, Interrupts, and Serial Ports

---

### COUNTER/TIMERS



**8051 family internal architecture**

As again shown, most of the 8051 family contain at least two *counters*. These are binary flip-flop chains. By programming one of the controlling special function registers (SFRs), you can configure I/O pins as inputs to the count chain. A counter, fed from the outside world, then increments on each high-low logic transition. With different programming of the SFRs, the in-

puts to a count chain comes from the internal oscillator/clock, and you then have a *timer*.<sup>1</sup>

The *counter* mode is usually for random external events. Traditionally, texts have talked about using counters with turnstiles in a subway station or with parts going down an assembly line. In reality, it is a waste of resources with so few internal counters to use them to track such slow events. You can easily keep track of slow events right in the software.<sup>2</sup> Use a counter for random events that may come more often than every 100  $\mu\text{sec}$ , and no more often than every 2  $\mu\text{sec}$ .<sup>3</sup>

The *timer* mode is best for measuring time between events or causing regular activities. Many microcontroller applications involve measuring the time between events or their frequency. For example, you can use a photo-emitter/detector pair to measure shaft rotation. With a slotted disc on a rotating shaft, you could get a pulse every time the slot allows light to pass from the emitter to the detector. With a single slot, the RPM of the shaft is the number of times the pulse arrives in a minute. Or, attach a few magnets to an automobile drive shaft, add a reed switch nearby, throw in a little math for the gear ratios and tire size, and you have a speedometer!

Likewise, a voltage to frequency (V-F) converter gives out a square wave with a frequency related to the signal from a sensor—temperature, humidity, pressure, wind speed, liquid level, light intensity, and so on.<sup>4</sup> Instead of using an (expensive) analog to digital (A-D) converter eating up ten to fourteen of the microcontroller's limited I/O pins, bring the frequency signal into the micro on a single line. By adding a digital multiplexing chip or a single-line tri-state buffer chip, you can even switch several different voltage-related signals on a single pin into one internal counter/timer. Frequency input is so much more economical for microcontrollers than Texas Instruments, for one,

---

<sup>1</sup>Whenever a counter increments at a *constant* rate, whether the signal comes from inside or outside the chip, it is properly termed a *timer*.

<sup>2</sup>Few real *physical/mechanical* events happen frequently enough to *require* one of the internal counters, let alone exceed its capability. A rare physical/mechanical event is too fast for the internal counter—to tax the counter's capability requires electronic inputs.

<sup>3</sup>Executing the software to watch a port bit, increment a counter, and loop back can easily take 10 or 20  $\mu\text{sec}$  and you could miss events. The internal counters work for events coming at least 2  $\mu\text{sec}$  apart. The internal counters only work for rates up to about 500 kHz because the count chains do not actually get the transitions directly. Instead, the input is *sampled* every twelfth internal clock cycle (1  $\mu\text{sec}$  for 12 MHz crystal). Thus the counter will miss brief pulses that come in between the sampling times.

<sup>4</sup>Chapter 14 develops an example deriving frequencies for both temperature and flow measurements.

has developed a family of photodetector devices that directly produce a frequency output proportional to the incoming light intensity.

### Internal Timer Details

The 8051 family devices have at least two internal timers. The 8052 relatives have three timers, and some of the newer devices have various mixes of additional devices.<sup>5</sup> The two basic 8051 timers are *T0* and *T1*. They are both 16-bit timer/counters with a variety of modes. Two special function registers (SFRs)—*TMOD* and *TCON*—control the modes of the internal timers.

If a timer is counting the internal crystal-driven clock, it is in the *timer* mode. If it is counting transitions on a designated input pin of the 8051, it is in the *counter* mode. The choice rests with the setting of the *TMOD* SFR. Two additional SFR pairs, *TH1/TH0* and *TL1/TL0*, determine the initial value in the timer. Bits in *TCON* turn on counting or timing.<sup>6</sup>

(msb)				(lsb)			
GATE	C/T	M1	M0	GATE	C/T	M1	M0
timer1				timer0			

#### **GATE**

0 ⇒ timer runs whenever TR0 (TR1) is set.

1 ⇒ timer runs only when INT0 (INT1) is high along with TR0 (TR1).

Note that use of external interrupt function is lost this way.

#### **C/T Counter/timer select**

0 ⇒ input from system clock (typically 1MHz)

1 ⇒ input from TX0 (TX1) pin. Note that a count input must be high or low for at least 1 microsecond and have a maximum frequency no more than 500kHz.

<sup>5</sup>See Appendix A4 for the listing of the 8051 family.

<sup>6</sup>Assemblers come with *include* files and C compilers come with *header* files that give names to and define the addresses of all the SFRs and bits of the particular 8051 family members.



**MODE 00** 13-bit counter lower 5 bits of TL0 (TL1) and all 8 bits of TH0 (TH1).

**MODE 01** 16-bit counter.

**MODE 10** 8-bit auto-reload. TH0 (TH1)  $\Rightarrow$  TL0 (TL1) when the latter overflows.

**MODE 11** (timer0) TL0 is 8-bit timer/counter controlled by timer0 control bits TH0 is 8-bit timer (only) controlled by timer1 control bits.

**MODE 11** (timer1) stops timer1 the same as setting TR1=0. (The most common use of mode 3 is to use timer1 as the baud rate generator and still have two 8-bit timers to generate interrupts.)

### Timer mode (TMOD) register

(msb)				(lsb)			
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

**TF0,TF1** Overflow flag—set by hardware when timer overflows. Cleared by hardware when processor vectors to interrupt routine.

**TR0,TR1** Run control bit set by software.

1  $\Rightarrow$  timer on.

0  $\Rightarrow$  timer off.

**IE0,IE1** Interrupt edge flag—set by hardware when external interrupt falling edge or low level is detected. Cleared when interrupt is processed.

**IT0,IT1** Interrupt type set by software.

1  $\Rightarrow$  falling edge triggered interrupt.

0  $\Rightarrow$  low level triggered interrupt

### Timer control (TCON) register

**FORGETTING THE TIMERS COUNT UP:** To get a timer to overflow after 100 counts start with:  $FF9C_{16}$  ( $-100$ ) so the counter gets to full count ( $FFFF_{16}$ ) after  $99_{10}$  increments. With the 8051 counters  $100_{10}$  will *not* decrement to 0 in 100 counts.

### Example: A 1msec Timer

The next program sets up *timer 0* as a 16-bit timer folding over to zero (an event that can trigger an interrupt) after 1000 clock cycles. In the *timer mode*, it counts the system clock.<sup>7</sup> I assume a 12 MHz crystal so a divide by  $1000_{10}$  gives a delay of 1 msec.<sup>8</sup> Because the counter counts *up*, it is necessary to load the timer with a value that will reach  $FFFF_{16}$  in  $999_{10}$  counts. Then the  $1000^{\text{th}}$  count will cause the count to go to 0000 (*roll over* or *fold-over*) and you can configure it to cause an interrupt. To get the *maximum* count to the roll over ( $65535_{10}$ ), load 0000 into the count registers.

---

#### Setting up to use Timer0

```
#include <reg51.h>
9  TMOD=0x01;
10 TH0=~(1000/256);
    TL0=-(1000%256);
    TR0=1;
```

---



---

#### Setting up to use Timer0

```
MOV TMOD, #00000001B
11 MOV TH0, #0FCH
    MOV TL0, #18H
    SETB TR0
```

---

<sup>7</sup>Typically this clock is 12 MHz always internally divided by  $12_{10}$  to give 1 MHz into the counter. For lower crystal frequencies the timer will increment less often and some of the newer family members can go up to 40 MHz.

<sup>8</sup>If the crystal were an 11.059 MHz one (common for proper function of the serial port—discussed later), to get a 1 msec delay would take a divide close to  $922_{10}$ . The algebra for the new divide is  $922_{10} \approx 1000_{10} * 11.059 \div 12_{10}$ .

<sup>9</sup>The Keil/Franklin C include file uses *upper case* for the register names and C is case sensitive here.

<sup>10</sup>The divide / and mod % are used in C to obtain the byte splitting. Most likely the level of optimization will cause the compiler to compute the constant in advance and not actually use the routines from the library at run time. The expression for *TH0* ( $-1000/256$ ) did not yield the expected  $FC_{16}$  but rather  $FD_{16}$  (the 2's complement).

<sup>11</sup>The  $FC_{16}$  is  $-1000_{10}$  in hex notation. With some assemblers, it could have been written out as an expression and the assembler would compute the value of the constant for you.

## Other Modes

If you are using a timer in the *counter mode*, the *C/T* bit of *TMOD* is different. A pin on *P3* becoming the counter input and, as with *RD* and *WR* for off-chip RAM, you lose the pin for normal port activity.<sup>12</sup>

For many on-going counting and time-duration operations, it is best to let the timers endlessly run as full-count devices. With no special software involvement, when the counter rolls over, it keeps on counting from 0000. After the initial setup of *TCON* and *TMOD*, the software only needs to read the value in the counter at the *start* of a count or time interval. At the *end* of the count or interval, the software can subtract a second reading from first. The *difference* is the count in between, or the duration of the interval.

For example, suppose you must measure the period of a signal from a V-F (voltage to frequency) converter. Assume the counter contents (running in timer mode) when a first logic 1 arrives from the V-F converter is  $3754_{10}$ . When the next logic 1 arrives, assume a count of  $4586_{10}$ . Then the time in between—the period of the V-F converter—must be  $832_{10}$  clock units or  $832_{10}$   $\mu\text{sec}$  (1.202 kHz).<sup>13</sup> The beauty of this approach is that there is no math problem if the counter should roll over as long as you use *unsigned* integer math.

An interesting mode is the *8-bit auto-reload* mode, in which the timer automatically restarts. Every time the count, *TLO*, reaches 0, the hardware automatically reloads it with the unchanging value in *TH0*. Any program that needs a regular interrupt at a period of 255 counts or less can thus avoid the software overhead (and time) to reload the counter at the end of the count.

## Timer2

If you have an 8052 or some of the other recent family additions, you have a *third* timer, *T2*, available for use. It is quite different to program and is much more flexible. *T2* includes a *16-bit auto-reload mode* so that the counter/timer will automatically short-cycle after it rolls over to 0 (like the

<sup>12</sup>While discussing lost pins, you can lose more pins to their normal port functions. You lose the upper 2 bits (6 and 7) of *P3* if you add external RAM (the *RD* and *WR* lines). For this (*T0*) counter input you lose bit 4. If the other timer serves as a counter you lose bit 5. You lose other *P3* bits (bits 0 and 1) for serial I/O. External interrupts use bits 2 and 3. Thus, *P3* is usually not a *full* 8-bit port except for very simple single-chip applications.

<sup>13</sup>If it is a 11.059 MHz crystal, that count of 832 would represent about  $903_{10}$   $\mu\text{sec}$  or 1.107 kHz.

8-bit auto-reload mode on timers 0 and 1). You can initiate the auto-reload by a transition on an external pin, *T2EX*. This external pin is useful for synchronizing the counter with other hardware. In this mode, the timer can also serve as a *watchdog timer* (time-out) where, if a pulse does not arrive within the timer period, it causes an interrupt to indicate that the pulse is late or missing.

(msb)							(lsb)
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL

**TF2** *Timer 2 overflow flag*—must be cleared by software.

**EXF2** *Timer 2 external flag*—set by negative transition on T2EX pin. EXEN2 is set (1). Indicates capture or reload has occurred and will vector to timer 2 interrupt if ET2in IE is set.

**RCLK**

- 1 ⇒ serial port receive clock comes from timer 2.
- 0 ⇒ timer 1 supplies baud clock.

**TCLK**

- 1 ⇒ serial port transmit clock comes from timer 2
- 0 ⇒ clock comes from timer 1

**EXEN2** *Timer 2 external enable flag*

- 0 ⇒ ignore pin T2EX.
- 1 ⇒ T2EX negative transition causes capture or reload. No effect if timer 2 supplies serial port clock.

**TR2** *Start/stop control*

- 1 ⇒ timer 2 runs.
- 0 ⇒ timer 2 stops.

**C/T2** *Counter/timer select*

- 1 ⇒ external event counter (pin T2).
- 0 ⇒ internal timer.

**CP/RL** *Capture/reload flag*

- 1 ⇒ capture on negative transitions of T2EX if EXEN2 is set.
- 0 ⇒ auto-reload when timer overflows or when T2EX makes negative transition. No effect if timer 2 is used as baud clock.

**Timer 2 control (T2CON) register**

*T2* also has a *capture mode*, which transfers the instantaneous count value to another register pair for reading by the processor. This avoids the danger of the counter value rippling between the 2 bytes during the reading process. This is useful for rapidly changing counts such as when you are measuring an external pulse width or period with the internal clock. If you read the *high* byte of the count when it is, for example,  $37FF_{16}$ , but it changes to  $3800_{16}$  before you can read the *low* byte, the (erroneous) result will look like  $3700_{16}$ . If you first *capture* the  $37FF_{16}$  into another 16-bit register, the computer can pick up the  $37_{16}$  and the  $FF_{16}$  at its leisure.

There are more details to the use of *T2* as well as a proliferation of other additions on 8051 relatives. These additions vary significantly although most of the relatives continue to support the basic 8051 functions.<sup>14</sup> There are numerous other registers for the various added features such as *T2* added the *T2CON* register. Consult the specific data sheets to go further.

## INTERRUPTS

An interrupt is something that stops what is happening to let something else happen. You can imagine eating dinner and being *interrupted* by a telephone call. The same thing can happen with the 8051 family of processors—a program can be running when some signal comes in that makes the processor switch over to running another program. For a computer, *an interrupt is a hardware-related event that stops the processor at the place in the program where it is running and causes it to move to some other place in the program.*

### How an Interrupt Works

An internal part of the hardware causes an interrupt by forcing a *CALL* into the instruction stream.<sup>15</sup> Like all calls, this pushes the program counter onto the stack and replaces it with the function's start address. Other than a few bit settings related to the hardware that caused the interrupt, that's all there is to it. The return can be with a normal *RET*, but there is a *RETI* instruction that resets one of the bits of the hardware at the same time.

---

<sup>14</sup>The '751 and '752 have only a single timer, but they have 16-bit auto reloads.

<sup>15</sup>Some of the less-integrated microprocessors used a separate chip for the interrupts (Intel's 8259 was common with  $\times 86$  processors). The principle is the same—an instruction that would not otherwise be there stops the current program instruction sequence in a way quite uncalled for by the current instructions.

The 8051-family interrupts are *vectored*—different interrupt events call different functions. Depending on which interrupt event occurs, the call is to the *fixed* address shown in the table. Either the C compiler or you, the assembly language programmer, must put jumps at *those* addresses to the actual location of the code you want to execute.<sup>16</sup> The routine you jump to when the interrupt comes is called an *interrupt service routine (ISR)*.

**8051-family hardware interrupts**

Name of interrupt		Vector Location
EX0	external 0	03H
ET0	timer 0	0BH
EX1	external 1	13H
ET1	timer 1	1BH
ES	serial	23H

Interrupt vectoring involves pushing the previous program counter value on to a stack (a *CALL*), and interrupt software (the interrupt service routine) ends much like any other routine (with a return). When an interrupt takes place in the 8051 family, the hardware disables all interrupts (the global *Enable All (EA)* bit of the *IE* register described shortly). As the vectoring takes place, the hardware clears the flag bit that indicated the interrupt source such as an internal timer or an external signal.<sup>17</sup> At the end of an interrupt function, the *RETI* instruction automatically re-enables the system to recognize other interrupts. It is not necessary to reset *EA* in the normal use of interrupts—just enable the interrupts once during the program initialization.

<sup>16</sup>These interrupt vector locations are spaced out at every 8 bytes so technically you could put your entire interrupt routine right there if it were no more than 8 bytes long, including the *RETI*. Since almost all interrupts involve a jump to somewhere else (only 3 bytes), some of the chips have an option to make the interval only 4 bytes.

<sup>17</sup>Automatic clearing does *not* happen for the flag bits for the serial port and timer2 (*ES* or *ET2*). Usually you want to determine in software if *RI* or *TI*, or *TF2* or *EXF2* caused the interrupt. Your interrupt service routine should branch different ways depending on the source of the interrupt. You will have to be careful to clear those various flags in software *before* re-enabling the global interrupt (*EA*). If you don't, the flag will cause an immediate repeat of the interrupt!

## Masking and Interrupt Enables

Whenever an application does not need some of the interrupts, you can *mask* them off with the *interrupt enable register (IE)*. Then those signals cannot cause interrupts. With the *external* interrupts, the pins involved also revert to their normal port functions. If you only unmask the interrupt from one timer, the other timer runs without causing an interrupt. If your program is at a critical place where you want to be immune to *all* interrupts, you can leave them all disabled with *EA*.

Normally, the interrupt hardware becomes sensitive to other interrupts only when you encountered the *RETI*.<sup>18</sup> Once set, an interrupt *request* flag (*IRQ*) will remain so until the software recognizes the interrupt. It is possible to software-initiate an interrupt routine by setting *TF0*, *TF1*, *IE0*, or *IE1* in *TCON*; *RI* or *TI* in *SCON*; or *TF2* or *EXF2* in *T2CON*.<sup>19</sup> Likewise, it is possible to *clear* interrupt request flags in software before they are recognized by the vector interrupt hardware (usually while they are masked or blocked by a higher or equal priority interrupt in progress).

(msb)								(lsb)
EA	—	ET2	ES	ET1	EX1	ET0	EX0	

### *EA*

0 ⇒ disables all interrupts.

1 ⇒ enables all unmasked interrupts.

### *ET2*

0 ⇒ disables timer 2 interrupts.

1 ⇒ enables timer 2 interrupts (overflow or capture).

<sup>18</sup>Actually, if you unmask in software you can let anything interrupt anything, but then you have to be careful about saving the *context*—discussed later. You can set *EA* directly in software at the start of the ISR to enable interrupts *before* the end of the routine. This is not recommended, however, since the interrupt could be interrupted by more interrupts including itself, possibly overflowing the stack and destroying registers. Any such ISR would have to be fully *reentrant*.

<sup>19</sup>This is a very common practice in the x86 family where there are specific software commands to cause interrupts. In DOS, it is the preferred way to call system services.

**ES**

0  $\Rightarrow$  disables serial port interrupts.

1  $\Rightarrow$  enables serial port interrupts (RI or TI).

**ETO,ETI**

0  $\Rightarrow$  disables timer overflow interrupt.

1  $\Rightarrow$  enables timer overflow interrupt.

**EXO,EXI**

0  $\Rightarrow$  disables external interrupt.

1  $\Rightarrow$  enables external Interrupt pin (INT0, INT1).

**Interrupt enable (IE) register****External Interrupt Hardware**

If you look back at *TCON*, you will see that half its bits are related to the external interrupts. *IT0* and *IT1* determine if the interrupt is edge-triggered—if set, then a negative going edge triggers a latch (*IE0* or *IE1*) that triggers the corresponding external interrupt. You can set or clear these two latches by software so, if the interrupt is not masked, you can *cause* an external interrupt by software. Likewise, if you had masked the interrupt and do not want past activity to trigger a new interrupt when you unmask, you can first clear the latches.

The internal hardware does not latch external *level*-triggered interrupts. If a *level*-triggered interrupt is already masked at the time it goes low and the level goes back high before the interrupt is unmasked, the level changes will be totally ignored.

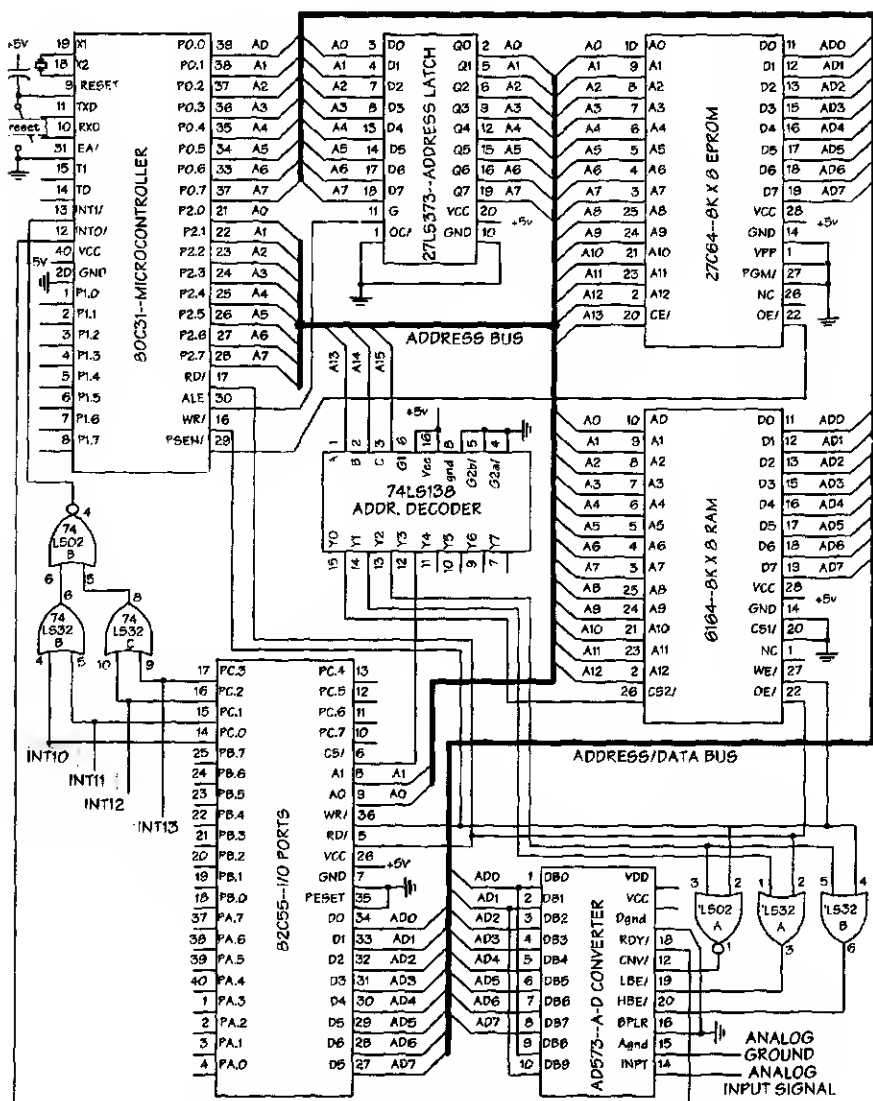
**Expanding Interrupts**

By adding external circuitry, you can *OR* multiple hardware interrupt sources into one of the two external interrupt pins. You then must *poll* (go around checking, by having the interrupt signals also tied in to port pins or by reading back a status register in some external device) to determine the specific source that caused the interrupt. With this approach, you do not have fully vectored interrupts, but it is more efficient than polling alone where there are many infrequently occurring interrupt sources.

The schematic that follows is an expansion of the last example in Chapter 2, using a few of the leftover gates to do wired-OR for four external interrupts. A high on any one of the four inputs gives the necessary low into the INT1/ pin. Once that happens, the program flow jumps to the interrupt



service routine (ISR) for interrupt 2 (vector address  $13_{16}$ ). With this hardware, you could read port C at the start of the ISR to see which pin(s) are high to determine which input(s) caused the interrupt.



**8031 Expansion with wired-OR circuit  
for four additional interrupts**

If you configure the 8051's external interrupt to be *edge* triggered, in order to recognize the transition from high to low that is sampled once each machine cycle, the input must stay low for at least one machine cycle—about 1  $\mu$ sec. The signal can then remain low indefinitely with no retriggering of an interrupt. For the next interrupt the signal must then stay high for at least one machine cycle before again going low. The interrupt flag (the flip-flop that is set by the transition) is automatically cleared when the interrupt is called, so no special software activity is needed to be ready for the next interrupt.<sup>20</sup>

If the interrupt is instead *level* triggered, the signal must stay low until the interrupt is generated—at least one machine cycle, but up to about 4  $\mu$ sec for a multiply—the level must be low when the current instruction ends. Then, if the interrupt is level driven, the signal must have gone away by the end of the interrupt service routine. Depending on what you do in the interrupt, this could be anywhere from a few tens of microseconds up through seconds. The point is that the level must have gone back high or the interrupt will start all over again.

If you need a *large* number of interrupts, it might be better to use open-collector gates tied together with a pull-up resistor. Notice that, even if the INT1/ pin is set to be edge triggered, *all* the interrupts must go low in order to have the next interrupt recognized. If you cannot guarantee that, then you have to add separate latches and clear signals to manage the interrupts. You could use edge-triggered latches by the interrupt signals and then clear them with an output bit of a port. The circuit complexity grows if the interrupts are not well behaved or if it is vital that you catch all interrupts.

### Interrupt Priorities and Latency

The 8051 directly supports only two levels of interrupt priority (*high* and *low* in the *IP* register) along with the background noninterrupt level, so use *different* priorities sparingly.<sup>21</sup>

---

<sup>20</sup>Be careful not to re-enable the global interrupt enable *EA* before the interrupt is done or you make the interrupt sensitive to being interrupted by itself. With the edge-triggered mode, it wouldn't happen before the next transitions high and then low, but with level triggered interrupts, if you re-enable *EA* while the level is still low you will immediately get a second interrupt. In other words, be very careful about resetting *EA* directly in an interrupt service routine—let the *RETI* that is put at the end take care of it for you.

<sup>21</sup>There is a way of getting a third priority, but it is not obvious and is not supported in C. Consult the "Architectural Overview" and the "Hardware Description" of the Intel data book to see these and other details from a different perspective. The software/hardware method of simulating a third priority level is described there. Incidentally, within a given priority level, the hardware priorities are (highest to lowest) *IE0*, *TF0*, *IE1*, *TF1*, *RI* + *TI*.

Unless you override it in software, when an interrupt is in progress, the hardware recognizes *no other interrupts* of the same or lower priority. The hardware normally masks low (0) priority interrupts when a high (1) priority or another low priority routine is running. It also masks high priority interrupts when any other high priority routine is running.

(msb)				(lsb)			
—	—	PT2	PS	PT1	PX1	PT0	PX0
<p><b>PT2 Timer 2 interrupt priority</b>  0 ⇒ low  1 ⇒ high</p> <p><b>PS Serial interrupt priority</b>  0 ⇒ low  1 ⇒ high</p> <p><b>PT1 Timer 1 interrupt priority</b>  0 ⇒ low  1 ⇒ high</p> <p><b>PX1 External interrupt 1 priority</b>  0 ⇒ low  1 ⇒ high</p> <p><b>PT0 Timer 0 interrupt priority</b>  0 ⇒ low  1 ⇒ high</p> <p><b>PX0 External interrupt 0 priority</b>  0 ⇒ low  1 ⇒ high</p>							

### Interrupt priority (IP) register

One measure of system response is called *interrupt latency* that expresses the *worst case* time from the arrival of an interrupt signal to the time when the interrupt software begins to execute, when no other interrupt is pending. At a minimum, the microcontroller must finish the currently executing instruction.<sup>22</sup> Worst case if it needs to finish

<sup>22</sup>Worst case is forty-eight clock cycles for a multiply.

a long interrupt service routine, the response can take quite some time.<sup>23</sup>

Avoid long ISRs if there are critical level-triggered external interrupt sources. In a long ISR, the system will miss an interrupt unless the hardware latches the request (as an IRQ). Even then, the controller cannot service new interrupt until the current one finishes.

## Context Switching

*Context switching is the activity involved in moving from one task to another when an interrupt occurs.* Register banks are the keys to doing it rapidly in the 8051 family.<sup>24</sup> Rather than saving a number of variables to a stack (normal with other processors), the simple change of two bits can shift operation to another set of eight registers. The software will not use the original set until it restores the bank-select bits—presumably when program flow returns to where it left off. In assembly, context switching is a matter of your choice.

For linking with mixed language programming, you can specify the banks used in the assembly program so the linker will not allocate the bank as ordinary memory. The control of register banks in C depends on the specific compiler. In Keil/Franklin, you can do it with the *using* directive as part of the function.

Since high priority interrupts can interrupt low priority ones in progress, pay attention to register banks for interrupt routines. Unless you can be sure *no* use is made of *R0* through *R7* (because you wrote the *assembly* code), assign different register banks to routines of each priority. Be sure that any routines used by the interrupt service routines also use the same register banks.<sup>25</sup>

---

<sup>23</sup>The measure of latency becomes a bit fuzzy if you include the time for an operating system to execute a context switch (discussed next), perhaps with a register bank change and pushing the accumulator and the B register. If you go on to get a measure of worst case response time when a previous interrupt has just happened, you have a number for the maximum possible delay before an interrupt can be acknowledged. For *that* measurement, the longer the other interrupt software subroutine/functions, the worse the number. For more on these sorts of system concerns, see the chapter on real-time operating systems.

<sup>24</sup>As already mentioned, the 8051 family has four register banks—groups of 8 bytes at the start of internal memory. The designations *R0*, . . . *R7* refer to a set of 8 bytes, depending on the setting of 2 bits in the *Program Status Word (PSW)*. Those bits decide at any given moment whether references to *R0-R7* will go to internal RAM address 0-7, 8-A, 10-18, or 18-1F.

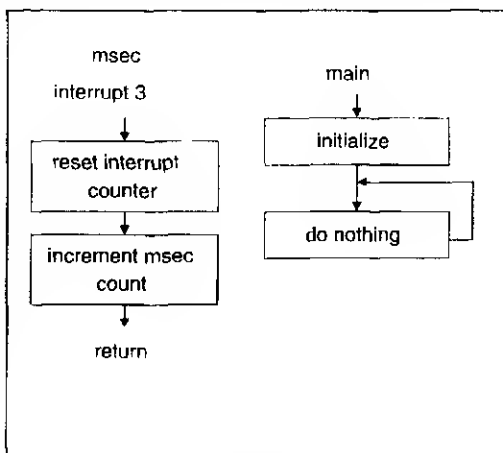
<sup>25</sup>Incidentally, Keil/Franklin's C compiler can specify register-independent functions and code (by using only direct addresses).

For faster, more efficient code, only have your interrupt service routines do very simple operations. Set flags so any lengthy processing can be done at the background (main program—noninterrupt) level. Save activity on longer variable types—especially floating-point math operations—for this background processing. For example, use an interrupt to bring raw serial characters into a buffer, but do line editing and command parsing later. Collect data and leave the averaging for later. This sort of design issue is an unavoidable part of efficient real-time systems.<sup>26</sup>

## Real-time Clock

In my opinion, the most important use of the internal 8051 hardware is in producing a *regular* interrupt. This regular interrupt is a *real-time clock*.<sup>27</sup> The clock “ticks” at a rate determined by the internal timer’s overflow, for example, every millisecond. Each overflow causes an interrupt.

A reasonable interval is about a millisecond for small hardware-oriented microcontroller systems and 10 to 100 msec for more complex, computation-oriented systems.<sup>28</sup> The next program sets up the timer and interrupt SFRs to create a regular interrupt. The real-time clock is the basis of virtually all multitasking operating systems.



Real-time clock flowchart

<sup>26</sup>This is discussed more in Chapter 13 (page 332) with real-time operating systems.

<sup>27</sup>Its function is described in Chapter 10, page 278.

<sup>28</sup>I hate to connect a real-time clock with tracking hours, minutes, and seconds because it will give you the idea that it is a clock to tell time. It *can* do that, but it is far more useful controlling the scanning of hardware inputs and the outputting of pulses to motors. On the other hand, properly arranged, the real-time clock can do *both* with no loss of accuracy. I find students want to add a separate clock chip to keep track of time even in applications where long-term accuracy and power-off retention are totally irrelevant and where the cost and hardware interface issues make the extra chip a poor economic choice.

---

```

                /*Real-time Clock*/
#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int
uint msec;

void msec(void) interrupt 3 using 1{
29    TH1=-1000>>8;
    TL1=-1000&0x00ff;
    msec++;
}

void main(void){
    TMOD=0x01;
    TH0=~(1000/256);
    TL0=-(1000%256);
    TR0=1;
    ET0=1;
30    for(;;);
}

```

---



---

```

                ;REAL-TIME CLOCK

MSEC SEGMENT DATA
MSECM SEGMENT CODE
MAINM SEGMENT CODE
RSEG MSEC
    DS 1
CSEG AT 0
    SJMP MAIN
CSEG AT 01BH
    INT3: SJMP MSEC
RSEG MSECM

31    MSEC: MOV TH1, #FCH ; INTERRUPT 3
        MOV TL1, #18H
        INC MSEC
        RETI

```

---

<sup>29</sup>This resets the timer for 1000<sub>10</sub>.

<sup>30</sup>Whatever you do in the background goes here. A halt could suffice.

<sup>31</sup>This resets the timer for 1000<sub>10</sub> so the interrupt comes every millisecond.

```

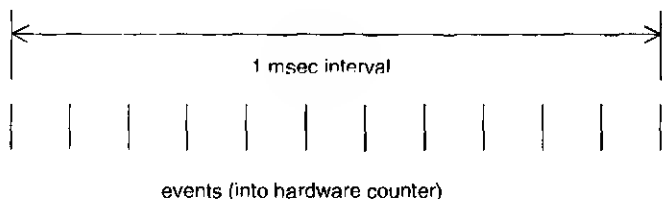
RSEG MAINM
MAIN:MOV TMOD,#1
      MOV TH0,#FCH
      MOV TL0,#18H
      SETB TR0
      SETB ET0
32    X:SJMP X
      END

```

---

## FAST EVENTS AND HIGH FREQUENCIES

You can count fast events, up to about 500 kHz (2  $\mu$ sec apart) in the micro's on-chip hardware counter. If there is a 1 msec real-time interrupt, read the count of external events (negative-going edges of the signal) at the start of every interrupt to get the count of events happening in a millisecond. For an incoming frequency of 100 kHz, at each interrupt the count will be 100.



For most ongoing counting and time-duration operations, do not reset the timers. When the counter rolls over, let it keep on counting. With *unsigned integer* math, subtracting a value obtained *before* the overflow from one *after* the overflow still gives the correct difference! If you read the value in the counter at the *start* of a time interval and then *subtract it* from the value at the *end*, you get the duration of the interval.<sup>33</sup> When the counter

---

<sup>32</sup>Whatever you do in the background goes here.

<sup>33</sup>Suppose you are using a timer to measure the duration of a signal from a V-F (voltage to frequency) converter. If the counter value when a logic 1 arrives is 3754, and the count when the next logic 1 arrives is 4586, then the period of the V-F converter is 832 clock units or 832  $\mu$ sec (1.202 kHz), with a 12 MHz crystal (if it had been with an 11.059 MHz crystal, that count would equal about 903  $\mu$ sec or 1.107 kHz).

rolls over, there is no math problem as long as you treat the readings as 16-bit *unsigned* integers.<sup>34</sup>

If you are measuring a 100 kHz signal, using a 1 msec interval you get a 1% resolution. If you need a higher resolution, let the counter accumulate for a longer time. If you subtracted the readings every 100th interrupt (100 msec), the 100 kHz could build up to a count of 10,000 or a resolution of .01%. That may be unnecessary accuracy, but it *would* give a 1% resolution for a 1 kHz frequency. Notice that this only gives ten readings a second. Somewhere in this frequency range, you could shift to measuring the *duration* of *one* pulse rather than counting pulses for a fixed duration. I discuss this next.

---

#### Measuring Fast Events

---

```
uint count;
void tracking (void) using 1{
    uint static newcount, oldcount;
    newcount=TH0<<8|TL0;
    count=newcount-oldcount;
    oldcount=newcount;
}
```

---

### Infrequent Events and Low Frequencies

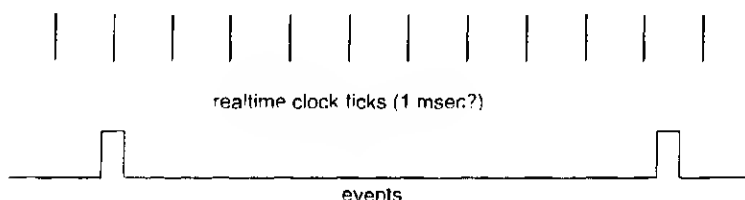
When events come at a rate of 10 Hz or less, it is usually better to write software to watch them at ordinary port lines. Check at the real-time clock interval (here every millisecond). In the interrupt, the software checks whether the incoming line has changed. When the line goes low, suppose it sets a variable to zero and clears a flag. Each real-time clock interrupt after that, the software increments the variable. At the first interrupt when the line is found to have gone high, it sets the flag. At the interrupt when the line is found low *and* the flag is set, the software transfers the variable's value to a "most recent period" variable and starts over. Any time the main program wishes to know the current reading, it can pick up the stored value at once.

---

<sup>34</sup>Some of the 8051 relatives have a 16-bit *capture register* to avoid the possibility of catching the two bytes of the count at different states. Otherwise reading the low-byte of 00000001 11111111 first could obtain 11111111 and, after another count and carry, reading the high-byte of 00000010 00000000, could result in a wrong result of 00000010 11111111. For slow count rates, this would happen very seldom, but if it is a major issue and the capture register is unavailable, your software could compare several readings to discard inconsistent ones.



Given a 1 msec interrupt, the reading will be the number of milliseconds between negative edges (the period) of the signal.




---

### Measuring Low frequencies

```
#declare PIN P1.0
uint count;

void tracking(void) using 1{
    uint static counting;
    sbit static flag;
    ++counting;
    if(PIN) flag=1;
    if(flag && !PIN){
        count=counting;
        flag=0;
        counting=0;
    }
}
```

---

If you want a number for frequency, you can invert the period, but steer away from introducing floating-point math just so you can have readings in a preconceived way. You can make decisions just as easily based on period as on frequency. Only if you need to display frequency for human consumption does the math make sense. Do not be chained to the units of your science classes and the approach of your pocket calculator for an embedded application that needs fast results with a minimum of processing. Even if you *must* have frequency, careful scaling may allow you to do the math in fixed-point form.

From 10 Hz down, a 1-msec interrupt will give at least a 1% resolution. If you measure *very* low frequencies, the count will overflow a 16-bit variable (in 65 s). To avoid overflow, before the increment you could test each interrupt to see if an overflow is imminent and carry to a second variable (or use a *long* in C). Alternately, you could have a counter that you

reset every 1000 interrupts, at which time you increment a *Seconds* variable. You can carry this process to *hours*, *days*, *weeks*, *months*, *years*, *centuries*, and so on.

### In-between Frequencies

If you have been alert, you realize that, for a 1% resolution, there is a band between 1 kHz and 10 Hz that is missing. The rate is too slow to measure in a reasonable time by counting and too fast to measure by counting the 1 msec interrupts. What are the choices?

First, you could speed up the interrupt to perhaps 100  $\mu$ sec. In this case, you accumulate the period in 100  $\mu$ sec units instead of 1 msec units. This might mean the overhead eats up most of the processor time—especially if there are many other things to do each interrupt.

Second, you can count events over a longer time and accumulate the frequency over a longer time. A 100 Hz signal can be resolved to 1% in 1 s. If you do not need frequent readings, this may be quite acceptable. It does not divert processor time from other activities. You can program the real-time clock to count 1000 or even 10,000 interrupts before reading the counter.

Third, the event could trigger an interrupt of its own. At the instant the interrupt comes, the internal timer can be read (remember it is counting at 1 MHz so it resolves to the nearest  $\mu$ sec). Since you already have a timer generating the 1 msec interrupt for the clock, just capture the current value in that hardware timer and combine it with the count being accumulated in the real-time clock's variable. Essentially you get a high-resolution time stamp for each event and can subtract to find the period. The math may not be trivial, but it is *doable*. In this case, you have freed up the second counter, but tied up one of the external interrupts.

### Broad-Range Frequencies

If you must handle a broad range of frequencies, I recommend the time stamp just described. Combining the counter reading with the real-time clock variables,<sup>35</sup> the interrupt can capture the exact instant to a resolution of 1  $\mu$ sec on out to eternity or to when the system is turned off, whichever comes first. With such a configuration, the hardware hookup does not change. You can even use external hardware to switch in various sources to the single interrupt for successive measurements of various frequencies.

---

<sup>35</sup>I assume you have variables that accumulate seconds, minutes, hours, and so on.

---

### Time Stamp Method

```

uint countusecs, countmsecs;

void event(void) interrupt 0 using 1 {
    uint static usecsold, msecsold;
    uint usecsnew;
    usecsnew = TH1 << 8 | TL1;
    countusecs = usecsnew - usecsold;
    countmsecs = msecsold - msecsold;
    usecsold = usecsnew;
    msecsold = msecsold;
}

```

---

## SERIAL PORTS: THE 8051'S UART

Most of the 8051 family members have on-board *universal asynchronous receiver/transmitters (UART)* for serial communication. If needed, you can do level shifting to RS-232 in a separate chip or to current loop with transistors.<sup>36</sup> Short serial links can use the TTL levels directly from the UART.

Common crystal and timer settings for 8051 UARTs

Baud Rate	fosc	SMOD	timer 1 mode	timer 1 reload value
19.2 K	11.059 MHz	1	2	FD <sub>16</sub>
9.6 K	11.059 MHz	0	2	FD <sub>16</sub>
4.8 K	11.059 MHz	0	2	FA <sub>16</sub>
2.4 K	11.059 MHz	0	2	F4 <sub>16</sub>
1.2 K	11.059 MHz	0	2	E8 <sub>16</sub>
137.5 K	11.086 MHz	0	2	1D <sub>16</sub>
110 K	6 MHz	0	2	72 <sub>16</sub>
110 K	12 MHz	0	1	FEED <sub>16</sub>

To determine the communication speed (*baud rate*) you must set one of the internal timers—timer 1 or, if available, timer 2. This is where the 11.059 MHz crystal comes into play. Since the necessary clock frequency

---

<sup>36</sup>RS-232, a standard for serial communication, is the most common communication mode for most computers and you can recognize it on a PC by the DB-9 or DB-25 connector on the back. The most common ICs for the level conversions are the MAXIM 232 family or with the 1488/89 chip set, both of which convert TTL to about  $\pm 10$  V to meet the specification. If your application requires the full handshaking features possible with RS-232 (for connecting to modems or recognizing if a cable is not connected) the features must be handled in software using additional port pins.

for the UART must come from the microcontroller's internal timers, the chosen timer must provide the proper frequency for the desired baud rate. From the table above you can see that the highest baud rates come out to an even division with a frequency just below the 12 MHz maximum.<sup>37</sup> Here are the registers for setting up the serial transmission, *SCON* and *PCON*.

(msb)						(lsb)	
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

### **SM0, SM1**

00  $\Rightarrow$  mode 0 Serial data exits and enters *RXD* pin while *TXD* outputs the shift clock. 8 data bits with lsb first and shift rate of 1/12 of osc. freq. (about 1 MHz).

01  $\Rightarrow$  mode 1 8-bit UART with baud rate determined by timer 1 or timer 2. Transmission is 10 bits: start bit (0), 8 bits from sbuf (lsb first) and a stop bit (1). The choice of timer 1 or timer 2 (only on 8052 members) is set by a bit in *T2CON*. This is the normal UART mode.

10  $\Rightarrow$  mode 2 9-bit UART with fixed baud rate. 11 bits transmitted/received including a ninth bit just before the stop bit. Baud rate is  $f_{osc}/64$  or  $f_{osc}/32$  depending on msb of *PCON* (see below). Ninth bit transmitted comes from th8 in *scon*. Ninth bit received goes into *TB8* of *SCON*.

11  $\Rightarrow$  mode 3 9-bit UART with variable baud rate as with mode 1. Same as mode 2 with respect to ninth bit.

### **SM2 Multiprocessor communication enable (modes 2,3)**

0  $\Rightarrow$  normal activation of *RI* when a character comes in.

1  $\Rightarrow$  *RI* only enabled if bit 9 (*RB8*) comes in high

### **REN**

1  $\Rightarrow$  enables serial reception. In mode 0 it sets the shift-in mode rather than the shift-out mode.

0  $\Rightarrow$  serial reception disabled. In mode 0 this sets the shift-out mode.

**TB8** The ninth data bit to be sent in modes 2 or 3.

<sup>37</sup>For very low rates there is enough leeway to use almost any crystal frequency.

**RR8** The ninth data bit that came in modes 2 or 3. In mode 1 it is the stop bit that was received.

**TI** Transmit interrupt flag. Set at the end of the eighth bit (mode 0) or the start of the stop bit by the hardware to allow the software to know when to load the next outgoing character. Must be cleared by software.

**RI** Receive interrupt flag. Set at the end of the eighth bit in mode 0 or half-way through the stop bit in any other mode (unless mode 2 or 3,  $SM2 = 1$  and the  $RR8 = 0$ ). Must be cleared by software.

### Serial port control register (SCON)

(msb)							(lsb)
SMOD	—	—	—	GF1	GF0	PD	ID

**SMOD** serial mode doubling bit

0  $\Rightarrow$  modes 1,3 baud rate = timer 1 overflow rate/32; mode 2 baud rate fosc/64

1  $\Rightarrow$  modes 1,3 baud rate = timer 1 overflow rate/16; mode 2 baud rate fosc/32

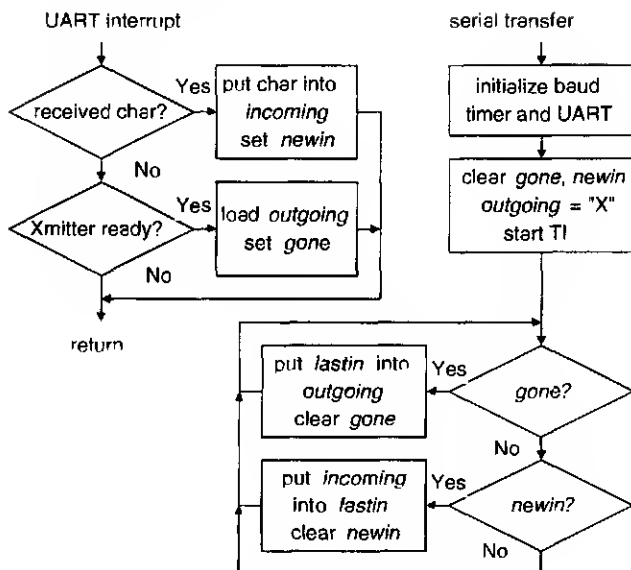
**GF0,GF1** Two general purpose bits useful as flags

**PD,IDL** Power down bits useful only on CHMOS devices. See Application notes such as Intel's *Hardware Description of the 8051, 8052 and 80C51* for more details on idle and power down modes.

### Power control register (PCON)

#### Example: Serial Buffering

The first example shows a typical initialization of the 8051 serial port for bi-directional communication with a terminal or another computer. There is initialization for 9600 baud, as well as an interrupt routine to handle characters in both directions. Without anything specific to do, I wrote it to send out the most recently received character endlessly (starting up with an X). There are many startup instructions in the main program to get the timer and UART ready. The interrupt function polls to determine if the *transmitter* or *receiver* caused the interrupt. In this simple program, the interrupt sets bit flags (*newin*, *gone*) and passes the characters back and forth with shared variables (*incoming* and *outgoing*).



Serial port usage flowchart

---

```

    /* Serial Port Usage Example*/
#include <reg51.h>
#define uchar unsigned char
uchar incoming,outgoing,lastinchar;
bit newin,gone;

38 void serint(void) interrupt 4 using 1{
    if (RI){
        incoming=SBUF;
        RI=0;
        newin=1;
    }
    else if (TI){
        SBUF=outgoing;
        TI=0;
        gone=1;
    }
}
  
```

---

<sup>38</sup>This is the serial port interface routine. Interrupt 4 is the *RI* or *TI* interrupt number. Note that it is necessary to test to see *which* signal caused the interrupt and that the flags hardware must be cleared by software.

```

void main(void){
    TMOD=0x20; /*timer 1 mode 2*/
    TH1=0xfd; /*9600 baud 11.059mhz*/
    TCON=0x40; /*start baud clock*/
    SCON=0x50; /*enable receive*/
    IE=0x90; /*enable serial int*/
    lastinchar=outgoing='X';
    gone=0;
    newin=0;
    TI=1; /*cause interrupt to start*/
    for(;;){
        if(gone){/*send latest char*/
            outgoing=lastinchar;
            gone=0;
        }
        if(newin){ /*catch latest char*/
            lastinchar=incoming;
            newin=0;
        }
    }
}

```

---

### **;SERIAL PORT USAGE EXAMPLE**

```

MAIN SEGMENT CODE
SERINT SEGMENT CODE
BYTEVARS SEGMENT DATA
BITVARS SEGMENT BIT
RSEG BYTEVARS
    INCOMING:DB 1
    OUTGOING:DB 1
RSEG BITVARS
    NEWIN:DBIT 1
    GONE:DBIT 1
CSEG AT 0
    JMP MAIN ;POWER-UP START
CSEG AT 023H
    JMP SERINT ;INTERRUPT4
RSEG SERINT
    SERIAL:SETB RS0 ;GO TO RBANK1
    JNB RI,PT2
    MOV INCOMING,SBUF
    CLR RI
    SETB NEWIN
    PT2:JNB TI,PT3

```

```

MOV SBUF,OUTGOING
CLR TI
SETB GONE
PT3:CLR RS0 ;BACK TO RBANK0
RETI

RSEG MAIN
START:MOV PSW,#0 ;SET RBANK0
MOV TMOD,#20H ;TIMER 1 MODE 2
MOV TH1,#FDH; ;9600 BAUD 11.059MHZ
MOV TCON,#40H ;START BAUD CLOCK
MOV SCON,#50H ;ENABLE RECEIVE
MOV IE,#90H; ;ENABLE SERIAL INT
MOV INCOMING,#'X';
MOV OUTGOING,INCOMING;
CLR GONE;
CLR NEWIN;
SETB TI ;CAUSE START
FOREVER:JNB GONE, PT4 ;SEND LATEST CHAR
MOV OUTGOING,INCOMING;
CLR GONE;
PT4:JNB NEWIN,FOREVER ;CATCH LATEST CHAR
CLR NEWIN;
JMP FOREVER
END

```

---

The second example, shown only in C, adds a first-in-first-out (FIFO) buffer function. The background program could “leave off” and “pick up” character strings in the buffers and the actual transfers to and from *SBUF* is done by the interrupt. The *loadmsg* routine loads the buffer array and signals the start of transmission. The buffers are each managed by two indices (*in* and *out*) and some flags. By making the buffers 32 bytes long, the indices can be managed by simple logical AND operations, which run more quickly than the modulus (%) operations. When *rbin=rbout*, then *rbuf* is full, and no more characters will be inserted. When *tbin=tbout*, then *tbuf* is empty, and the transmit interrupt will be cleared to stop further requests by the UART. Other approaches to first-in-first-out buffers are possible—focus here on the serial interrupt routine. I omit the assembly language version because the details would just obscure the overall approach.

---

```

/*Serial Port Buffer Example*/
#include <reg51.h>
#define uchar unsigned char

```



```

39  uchar xdata rbuf[32];
    uchar xdata tbuf[32];

40  uchar rbin,rbout,tbin,tbout;
    bit rfull,empty,tdone;
    uchar code m1[]={"this is a test\r\n"};

41  void serint(void) interrupt 4 using 1{
    if (RI && ~rfull){
        rbuf[rbin]=SBUF;
        RI=0;
        rbin=++rbin & 0x1f;
        if (rbin==rbout) rfull=1;
    }
    else if (TI && ~empty){
        SBUF=tbuf[tbout];
        TI=0;
        tbout=++tbout & 0x1f;
        if (tbout==tbin) empty=1;
    }
    else if (TI){
        TI=0;
        tdone=1;
    }
}

42  void loadmsg (uchar code *msgchar){
    while ((*msgchar!=0) &&
           (((tbin+1)^tbout)& 0x1f)!=0)){
        /*test for buffer full*/
        tbuf[tbin]=*msgchar;
        msgchar++;
        tbin=++tbin & 0x1f;
    }
}

```

---

<sup>39</sup>These are the buffers where background tasks leave off strings for transmission or pick up incoming strings.

<sup>40</sup>These pointers keep track of where in the buffers characters are going in or coming out.

<sup>41</sup>This is the serial port interface routine much like the previous example. The full/empty flags keep characters from overflowing the buffers. The pointers are set up to use equality (which is quick to test) within the interrupt routine while the "increment-and-fold around" testing is done in the (non-interrupt) background. This sort of thinking is part of managing interrupt-driven activities.

<sup>42</sup>This function puts a character string into the transmit buffer. It retrieves only out of the *code* space—a separate function or a generic 3-byte pointer would be needed to pull strings out of *xdata* space.

```

43         if (tdone){
                TI=1; tempty=tdone=0;
                                /*restart xmit if all finished*/
        }
    }
}

44 void process (uchar ch){return;}
                                /*who knows what?*/

void processmsg(void){
    while (((rbout+1)^ rbin)!=0){ /*not empty*/
        process(rbuf[rbout]);
        rbout=++rbout & 0x1f;
    }
}

void main(void){
    TMOD=0x20; /*timer 1 mode 2*/
    TH1=0xfd; /*9600 baud 11.059mhz*/
    TCON=0x40; /*start baud clock*/
    SCON=0x50; /*enable receive*/
    IE=0x90; /*enable serial int*/
    tempty=tdone=1;
    rfull=0;
    rbout=tbin=tbout=0;
    rbin=1; /*rbuf and tbuf empty*/
    for(;;){
45         loadmsg(&m1);
        processmsg();
    }
}

```

---

<sup>43</sup>If transmission has previously finished, it is necessary to manually set *TI* to restart the interrupt routine for new transmissions.

<sup>44</sup>This null routine (just a return to simplify the simulation) could parse strings and make decisions based on incoming characters.

<sup>45</sup>The actual main program just loads the test message over and over and unloads any incoming characters. It is obviously not a final application.

## Shift Register Mode

The serial port has two sometimes-overlooked modes. There is a *shift register mode*, which is useful for simple I/O expansion, as well as communication and a *ninth-bit mode*, which is quite powerful for dedicated interconnection of processors.<sup>46</sup>

The shift register mode is useful for expanding ports with a minimum of hardware because a simple 8-bit shift register can be loaded directly for output or shifted in for input. Clocking at 1 MHz (a 12 MHz crystal), the 8 bits load in about 10  $\mu$ sec. Any number of shift registers can be cascaded for a large number of I/O bits. This could be a very economical system of I/O expansion if the ripple during the shifting is not important or if parallel-load latches are included with the shift registers.

A second use of the shift register mode is for communication between two processors. For short distances, the ability to communicate at 1 MHz is quite impressive when compared with normal 9600-baud serial communication. Because the same port pins are involved for either direction, it is possible, with a few additional gates, to add communication among *several* processors. A bit or two from the normal parallel ports can set up connections and directions. Clearly, there is no standard for such connections, and the system must be custom-designed for each application.

## Ninth-bit Mode

An unusual feature of the 8051 serial port is the *ninth-bit mode*. This inserts an extra bit in the serial transmission that you can use to flag the receiver for special characters. For a simple master-slave network, the ninth-bit scheme allows you to interrupt multiple receiving controllers only by characters having a one as an extra (ninth) bit. In that way, the transmitter can broadcast a byte with the ninth bit high as the *everybody pay attention* byte. The byte could hold the address of the receiver that should stay on for the following characters. All the following bytes (with the ninth bit low) would cause no interrupt to the other receivers, because they would have shut themselves off (disabled their receive interrupts). In this way, one microcontroller could talk to any number of other controllers without interrupting controllers that you did not address. Replies to the master could work as wired-OR connections if the signals remained at TTL levels.<sup>47</sup> From a net-

---

<sup>46</sup>Some chips also have an I<sup>2</sup>C bus described in the companion book.

<sup>47</sup>It is not so simple with RS-232, because multiple transmitters on the same line would conflict. An arrangement with RS-422 or RS-485 could work.

working sense, this would be quite primitive, with no collision detection and such a system would have to operate in a strict master-slave fashion or with a token-passing arrangement. Such ground-up designs rapidly lead to very nonstandard systems.

## REVIEW AND BEYOND

1. How many priorities are inherent in the 8051 interrupts?
2. What is the difference between a timer mode and a counter mode?
3. What provides the timing for serial communication with the 8051 family?
4. Why does the program example initialize the counter with a “negative” value? Could you avoid this? Explain.
5. In the serial buffering example, could you get into trouble if both RI and TI were set before the interrupt routine started? How could you change the function to fix this (if it is a problem)?
6. What is a real-time clock?
7. What is the difference between an edge-triggered and a level-triggered external interrupt?
8. Explain how you could handle external interrupts from, for example, twenty different switches.
9. Explain the problems in measuring very slow events.

# 12

## Build Your Own Scheduler

---

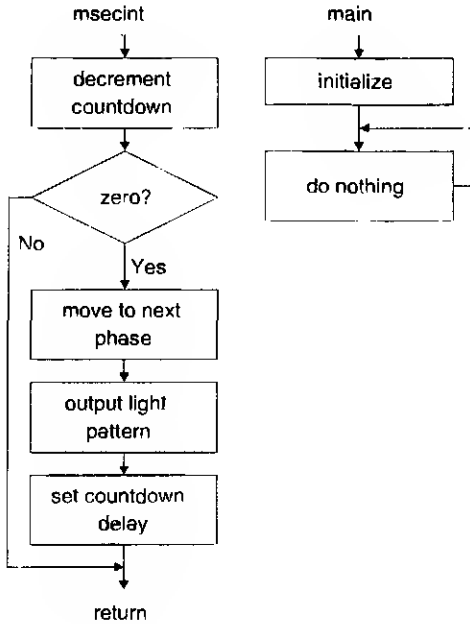
This chapter develops the simplest of real-time systems—the scheduler. It is a long way from the real-time operating systems of the next chapter, covered more in the companion book, but the principles are the most useful ones for making efficient programs with microcontrollers. Although you will first find a scheduler to handle traffic lights—something done with 4-bit microcontrollers thirty years ago—do not imagine that the concepts are obsolete. Further examples deal with solenoids, a pulse generator, and an envelope detector.

Having shown a real-time interrupt (real-time “clock” if you prefer) in the last chapter, we come to the question of what to *do* with it. Perhaps the simplest form of multitasking uses this interrupt to make a *scheduler*—software that runs tasks at specific times on a *schedule*. In the two examples that follow, the scheduler runs the cycle of a traffic light. This is an appropriate use of a scheduler because the changes of the light should happen at specific, regular times no matter what else the processor is doing. Having a clock “ticking” at 1 kHz, the main program can ignore time management. Things that happen unrelated to the activities of the real-time clock’s interrupt (such as pushing a walk request button) can change the schedule, but all the state changes of the lights happen at system ticks (real-time clock interrupts).

### EXAMPLE: TRAFFIC LIGHT (BASIC CYCLE)

Let us first make a simple traffic light and then later add two buttons so pedestrians can request a *walk* cycle. We can handle the light changes in the

real-time interrupt, and then handle the requests for a *walk* by polling in the main routine.<sup>1</sup> A normal traffic light cycle consists of sending out the light pattern and then waiting the specified number of interrupts before changing to the next pattern.<sup>2</sup>



**Traffic light flowchart—basic cycle**

#### Basic Traffic Light

```

#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int
#define LEDS P1
struct {uint delay;uchar pattern;}cycle[]={
    {{1000,0x12},{3000,0x41},
    {1000,0x21},{5000,0x14}}};
/*-R,-Y-;G-,-R;-Y-,-R;-R,G-+ /

```

<sup>1</sup>I could handle the *walk* requests with separate hardware interrupts, but this example, instead, shows communication between the interrupt activities and the main (background) program.

<sup>2</sup>I do not show the hardware because, for simulation purposes, it is just a few LEDs tied to P1.

```

void msec(void) interrupt 3 using 1{
    static uchar index;
    static uint countdown;
    TH1=-1000>>8;
    TL1=-1000&0x00ff;
    countdown=countdown-1;

    if(countdown==0){
        index=index+1;
        if(index>3)index=0;
        LEDS=cycle[index].pattern;
        countdown=cycle[index].delay;
    }
}

void main(void){
    TMOD=0x10;
    TH1=~(1000/255);
    TL1=~(1000%255);
    TR1=1;
    IE=0x11;
    for(;;){

```

---

### **;TRAFFIC LIGHT PROGRAM<sup>5</sup>**

```

VARS SEGMENT DATA
TBL SEGMENT CODE
TRAFFIC SEGMENT CODE
RSEG VARS
    index:DS 1
    countdown:DS 2
PESG TBL
    cycle:DW 1000
        DB 12H
        DW 3000
        DB 41H
        DW 1000
        DB 21H
        DW 5000

```

---

<sup>3</sup>Update countdown at end of delay.

<sup>4</sup>Nothing to do in main yet.

<sup>5</sup>This is the version generated by the C compiler, with the variable definitions added. I omit the next version in assembly—things are getting too big to be very useful in grasping assembly. It is a very useful exercise to see how the C compiler executes the instructions. Can you do better?

```

        DB 14H
RSEG TRAFFIC
        ;FUNCTION main (BEGIN)
        MOV TMOD,#010H
        MOV TH1,#0FCH
        MOV TL1,#018H
        SETB TR1
        MOV IE,#011H
?C04:SJMP ?C04
        ;FUNCTION main (END)
        ;FUNCTION msec (BEGIN)
        PUSH ACC
        PUSH B
        PUSH PSW
        MOV PSW,#08H
        MOV TH1,#0FCH
        MOV TL1,#018H
        MOV A,countdown+01H
        DEC countdown+01H
        JNZ ?C07
        DEC countdown
?C07:MOV  A,countdown+01H
        ORL A,countdown
        JNZ ?C03
        INC index
        MOV A,index
        SETB C
        SUBB A,#03H
        JC ?C02
        CLR A
        MOV index,A
?C02:MOV A,index
        MOV B,#03H
        MUL AB
        ADD A,#cycle+02H
        MOV R0,A
        MOV A,@R0
        MOV P1,A
        MOV A,index
        MOV B,#03H
        MUL AB
        ADD A,#cycle
        MOV R0,A
        MOV A,@R0
        MOV R6,A
        INC R0

```



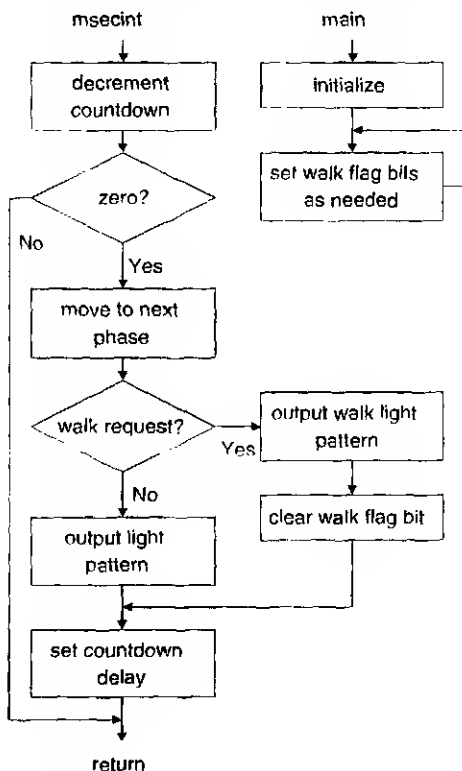
```

MOV A, @R0
MOV countdown, R6
MOV countdown+01H, A
PC03: POP PSW
POP B
POP ACC
RETI
;FUNCTION msec (END)

```

## EXAMPLE: TRAFFIC LIGHT (WITH WALK BUTTONS)

So far, nothing is happening in the main program. It is an empty *forever* loop. Let's revise the program to have two walk request buttons polled in the main program. Set up a flag (*walk*) to share between the main program and the interrupt. To be shared it will have to be global—declared outside the functions. The walk lights will be the msb of each nibble of the pattern going to the LEDs. Notice in the flow chart that the interrupt is not part of the main program flow—always the case with an interrupt function.



**Traffic light flowchart—with walk buttons**

**Traffic Light With Walk Buttons**

```

#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int
#define LEDS P1
#define PBS P3
uchar walk=0;
struct {uint delay,uchar pattern}cycle[]=
    /*--R,-Y-;      -G-,--R;      -Y-,--R*/
    {{1000,0x12},{3000,0x41},{1000,0x21},
    /*--R,-G-;      WG-,--R;      --R,WG-*/*
    {5000,0x14},{3000,0xc1},{5000,0x1c}};

void initialize(void){
    TMOD=0x10;
    TH1=~(1000/255);
    TL1=~(1000%255);
    TR1=1;
    IE=0x11;
}

void msecint (void) interrupt 3 using 1{
    static uchar index;
    static uint countdown;
    uchar i;
    countdown=countdown-1;
    if((countdown==0){ /*update at end of delay*/
        index=index+1;
        if(index==4) index=0;
        switch(index){
            case 1:if(walk&0x2>0){
                walk=walk & 0xfd;
                i=index+3;
            }
            break;

```

---

<sup>6</sup>Set index to next state. Could have used `index=++index^4`;

<sup>7</sup>One of the features of C is the definition of *true* and *false*. Anything other than zero is a value for *true*. Only zero is a value for *false*. In the line below, the *true* condition occurs when the bit is set high. It could have been written as `if(walk&0x02>0)`; or `if((walk&0x2)==2)`; Note that the `&` grouping does not have to have the extra parenthesis because the `>` or `==` has higher precedence than the `&`. Be careful! If in doubt, always use extra parenthesis.

<sup>8</sup>Reset walk request flag. Could have used `walk&=0xfd`;

<sup>9</sup>Use alternate light pattern for walk cycle.

```

        case 3: if (walk & 0x1 > 0) {
            walk = walk & 0xfe;10
            i = index + 2;
        }
        break;
        default: i = index;
    }
    LEDS = cycle[i].pattern;
    countdown = cycle[i].delay;
}

void main(void) {
    initialize();
12 for(;;) walk = walk | PBS;
}

```

---

## MORE ELABORATE COMMUNICATION

More elaborate operating systems pass information by copying actual data or transferring pointers to blocks of information. Some systems create a *semaphore*, which can be just like a flag, or can be a *counting* semaphore. In the latter case, each time you add a *unit* the count goes up by one and each time you remove a unit the count goes down. Suppose you had a task that should not run until three stepper motors had gotten home. You could put three *units* in a semaphore and have each stepper task remove one unit when done. The task to run when all the stepping finishes could be inactive until the semaphore units got to zero. With a scheduler, you could have the checking of the units in the semaphore an automatic part of each tick. That way the three stepper tasks could finish in any order and the follow-on task would only run when all three finished.

## BREAK-OUT FUNCTIONS

If you have multiple things to process in the scheduler interrupt, bring out the individual parts in separate functions. Bringing out the body of the cycle processing makes it necessary to *return* the next value of *countdown* to the

---

<sup>10</sup>Reset walk request flag.

<sup>11</sup>Use alternate light pattern for walk cycle.

<sup>12</sup>Set bits for walk requests.

interrupt from the function because the *countdown* value is private to the interrupt. The goal in your main program is to have the interrupt function easy for the casual reader to follow.

---

### Traffic Light With Called Function<sup>13</sup>

```

uint trafficcycle(void) using 1{
    static uchar index;
    uchar i;
14    switch(index=++index%4){
        case 1:if(walk&0x2){
15            walk&=0xfd;
16            i=index+3;
            break;
        }
17        case 3:if(walk&0x1){
            walk&=0xfe;
18            i=index+2;
            break;
        }
        default:i=index;
    }
    LEDS=cycle[i].pattern;
    return cycle[i].delay;
}

void msecint (void) interrupt 3 using 1{
    static uint countdown;
19    if(--countdown==0) countdown=trafficcycle();
}

void main(void){
    initialize();
20    for(;;) walk=walk | PBS;
}

```

---

<sup>13</sup>The same starting instructions would go here as in the two examples early in this chapter.

<sup>14</sup>Set index to next.

<sup>15</sup>Reset walk request.

<sup>16</sup>Alternate for walk.

<sup>17</sup>Reset walk request.

<sup>18</sup>Alternate for walk.

<sup>19</sup>Update at end of delay.

<sup>20</sup>Set bits for walk request.

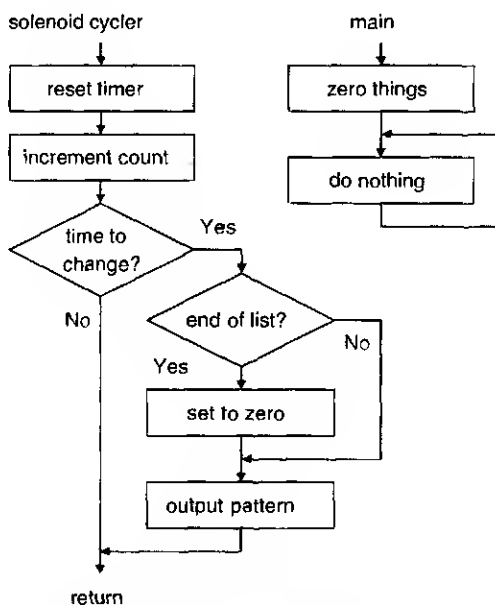
## COMMUNICATION: SHARED VARIABLES

In the previous example, the interrupt and the main program share the information about the walk requests. A global *shared variable* is the easiest way to coordinate or exchange information between tasks. Set a flag in one task to *request* processing and then clear it in another task to *acknowledge* the completion of processing. You can exchange not just information represented by flags, but also numbers, pointers to messages, or even messages themselves. This is where commercial operating systems shine, but they utilize more elaborate versions of the concepts presented here.

## NOTHING TO DO

Although the interrupt in the stoplight example has grown more complicated, the basic approach has not changed. The program flow never waits in the interrupt function. Rather it moves through and returns to the main as quickly as possible. The program, as it stands, is polling the walk buttons at a fierce rate. *main* has nothing else to do and stops only briefly for each millisecond interrupt.

Once you take the timing out of the hands of software delay loops, you will find that seldom will your programs tax the capability of the microcontroller.

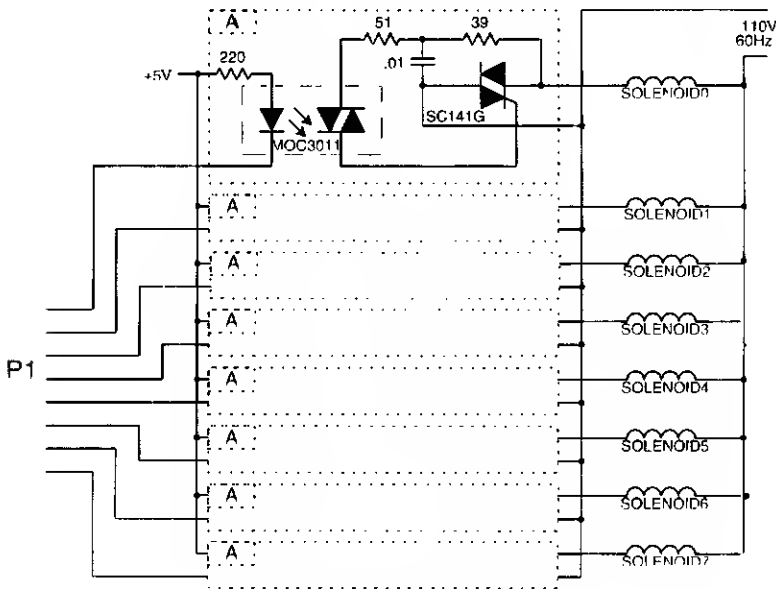


**Solenoid cyclor flow chart**

## EXAMPLE: SOLENOID CYCLER

For another example, consider a scheduler that turns on and off two solenoids attached to bits 0 and 1 of *P1* in a fixed time pattern. Designate the solenoids as *S1* and *S2*. At the cycle start they are both *OFF*. Two seconds later *S1* comes *ON*. A 1/10 s later *S2* comes *ON*. *S1* stays *ON* for 2.0 s and *S2* stays *ON* for 2.4 s. For this example, the total cycle takes 4.5 s. A dynamically assigned cycle would be more realistic, with user inputs for setup, but this is simpler for a start. This example differs from conventional programming because it uses a real-time interrupt. The match with the counter incremented in the real-time interrupt determines the intervals.

The schematic for a solenoid driver circuit shows the use of an opto-isolated triac driver with a separate triac to drive the solenoid coil. Industrial solenoids usually have 110 vAC (or 220 v) coils, and it is best to include optical isolation. If something fails at the output, you do not want the high voltage to destroy the entire controller or get back the user panels and switches. In addition, opto-isolation should improve noise immunity.



**Solenoid cyclers schematic**

### Solenoid Cyclers-C

```
#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int
```

```

    uchar i;
    uint nowtime;

21 struct code{uint abstime;uchar pattern;}next []=
        {{0,0x00},{200,0x01},{210,0x03},{400,0x02},
        {450,0xff}};
22 void cycletimer(void)interrupt 1 using 1{23
24     TH0 = -8333/256;
        TL0 = -8333*256;
        ET0 = 1; 25

26     nowtime++;
        if (nowtime==next[i].abstime){
27         if (next[i].pattern!=0xff) i=nowtime=0;
            P1=next[i++].pattern;
        }
    }

    void main(void){
        nowtime=i=0;

28        for(;;):
    }

```

---

## EXAMPLE: PULSE GENERATOR

The next example brings about a regular, variable pulse width signal that you can use to drive a motor or as a "poor man's D-A converter."<sup>28</sup>

---

<sup>21</sup>This structure is a fixed-content schedule for the scheduler. The first part of each structure element is the absolute time at which the corresponding solenoid states are sent out. Here only two solenoids are in use.

<sup>22</sup>This interrupt is the key to the scheduler. Every 10 msec the interrupt causes this interrupt routine to run. The first step is to reset the timer and start it out again so the minimum of time is lost before the next time interval is started.

<sup>23</sup>Real-time interrupt routine happens every 10 msec.

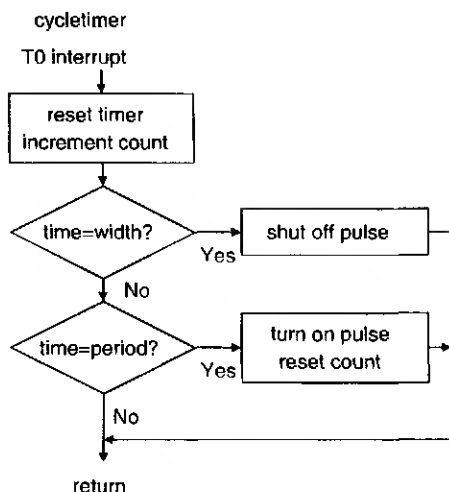
<sup>24</sup>10 msec with 10 MHz crystal.

<sup>25</sup>This variable is the counter of 10 msec units into the cycle.

<sup>26</sup>A 0xff for the pattern indicates the end of the cycle.

<sup>27</sup>Endless do-nothing background task.

<sup>28</sup>Although you cannot change the values of *width* and *frequency* in this example, for a finished design you would probably change them from outside by some user-input task using shared variables, signaling, or messages (Chapter 13).

**Pulse task flowchart**

---

**Pulse Width-C**

```

#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int
uchar i; uint nowtime;

29  uchar width=50;
30  uchar freq=250;
31  void cycletimer(void) interrupt 1 using 1 {
32      TH0 = -8333/256;
      TL0 = -8333%256;
      ET0 = 1; 31
33      if (++nowtime==width) P1=0;
      else if (nowtime==freq){
          nowtime=0;
          P1=1;
  
```

---

<sup>29</sup>20% duty cycle—50 out of 250.<sup>30</sup>These variables are set to initial values so the program will do something on startup.<sup>31</sup>This real-time interrupt routine happens every 10 msec.<sup>32</sup>10 msec—10 MHz crystal.<sup>33</sup>*Nowtime* is incremented *before* the test. If the time reaches the number in *width* the high part of the pulse ends, while, if the time count reaches *freq*, a new pulse starts.



```

    }
}

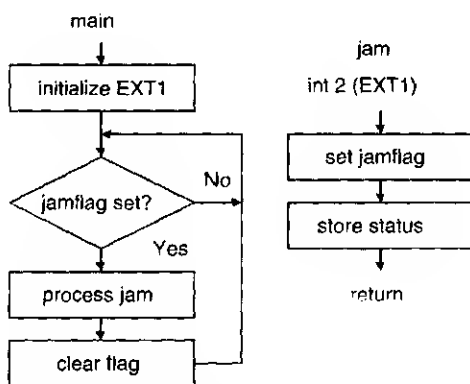
void main(void){
    forever{

```

---

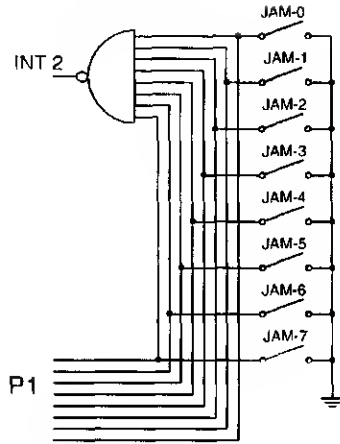
## EXAMPLE: ENVELOPE DETECTOR

Take an envelope-handler system with numerous sensors along the path of the conveyor and stuffer systems. After an interrupt the microcontroller could either poll the sensors or use some sort of *wired-OR* to bring the sensor signals into a single hardware interrupt. The schematic shows a possible system for bringing in multiple switch or photo-interrupter signals to a single interrupt using a NAND gate and a port. Alternately, a 74148 priority encoder chip could do the job using less port pins.



**Envelope detector—flowchart**

This interrupt will only report the particular sensor causing the interrupt (from polling) to another task (for further processing).



Envelope detector—schematic

## Envelope detector

```

#include <reg51.h>
#define uchar unsigned char
#define uint unsigned
uchar status;
bit jamflg;

void jam (void) interrupt 2 using 2 {
    jamflg=1;
    status=P1;
}

void main (void){
    uchar jamnumber;
34  IP=0x04;
35  IE=0x86;
    for(;;)
        if(jamflg){
36      for(jamnumber=0;status!=0;
           jamnumber++,status>>=1){
           if(status&1==1){

```

<sup>34</sup>Set IE1 high priority.<sup>35</sup>Enable IE1 and IT0.<sup>36</sup>This *for* loop illustrates the many possibilities. Here *jamnumber* is initialized to zero, the looping continues until all the jam bits are processed, and the *jamnumber* is incremented each time *status* is tested for one more bit—here is a use for the comma.

```

        switch (jamnumber){
            case 0: break;
            case 1: break;
            case 2: break;
            case 3: break;
            case 4: break;
            case 5: break;
            case 6: break;
            case 7: break;
            default:;
        }
    }
    jamflag=0;
}
}
}

```

## DELAYED TICK DUE TO OVERLOAD

If there is a lot to do during the system tick, the processing might go on beyond the next tick. At that point, there would be no time before the next interrupt. As configured up to now, any activity in the interrupt requiring longer than 1 msec will still run without interruption. In the 8051 family at least, the processor hardware automatically *disables* all interrupts until the current interrupt routine is exited. Re-enabling the interrupt is automatic with a *RETI* instruction. The hardware would delay the *next* interrupt: when the first, overly long interrupt finished, it would lead immediately into the start of the second interrupt. In other words, the system would delay the second interrupt but would allow it to start when the first one finishes. As long as the system never got *two* ticks behind, all the ticks would happen and things would get back on time quickly. If it got two ticks behind, the second timer overflow would set a flip-flop that it already set before, and you would lose one tick.

---

<sup>32</sup>Handle jams here.

## CATCHING ALL TICKS

If you *reenabled* the interrupts right at the *start* of the interrupt then the interrupt could interrupt itself! That way you could process successive ticks before the long processing of the first tick had finished. If you do this, however, be very careful. Remember that you may end up processing the variables in reverse time order and, if you do not put them on the stack, the processor may overwrite them. The interrupt function must be *reentrant* so that the second invocation does not mess up something going on in the first invocation. At a minimum, to be re-entrant, the function must have no static internal variables and should affect no global variables.

## REVIEW AND BEYOND

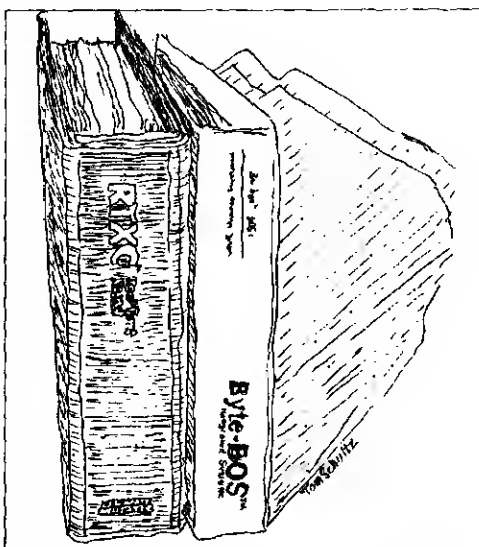
1. What happens if you use a time-burning delay loop with a multitasking system? Why is it not a good idea?
2. What is the difference between a signal and a message? Can you use them interchangeably?
3. When would a shared variable be preferable to messages or signaling? What are the advantages of the latter?
4. What is the difference between passing message *contents* and only passing a message *pointer*? What cautions are necessary with the latter? What costs are associated with the former?

# 13

## Real-time Operating Systems

---

So far the multitasking has all been of the “write your own” variety. If you are doing a large project where you work with other programmers, or if you like a more formalized approach, you can use an *operating system* to handle the details of managing separate tasks. Such systems are not magic—they use the same resources that you have as a programmer, but they can standardize your software—very valuable when someone else comes along later to modify your code. You have used operating systems such as



DOS or Windows® on a PC to simplify disk access or graphical functions. In this brief chapter, I only give you an overview of multitasking operating systems.<sup>1</sup>

---

<sup>1</sup>The companion book goes into much more detail with both write-your-own and commercial systems.

## WHY AN OPERATING SYSTEM FOR MULTITASKING?

Why might you want to use a full multitasking operating system? First, multitasking helps you avoid some of the programming headaches that come with developing embedded control applications. The system calls formalize and standardize things, making it easier to include multitasking. Your programs become more readable to someone else and the operating system defines the interactions more consistently.

Second, unlike many operating systems for other processors, *some* of the 8051 operating systems are very small and control-oriented. They handle multiple real-time inputs and outputs very quickly and efficiently. You do not give up as much as you might expect by using one instead of writing your own. Not all the operating systems fit the 8051 well—some were originally for other processors that are more stack-oriented. If these others are written in C, it is easy to port over to the 8051, but that does not make an efficient system. You can approach context switching in many ways that have a big impact on size and speed.

Finally, if your projects grow and change as much as mine have, with an operating system you will not have to reconsider and scrap large pieces as the “creeping features” come along. You will not have the task interactions buried in the subtle, clever (and obscure!) approach you carefully developed from scratch at the beginning. At the same time, your programming should be portable to some other higher-performance processor if necessary. In the debate *between* operating systems, some argue that “pure” ANSI C should be easier to port to a different processor. You have to weigh speed and efficiency now against code portability in the future.

## HOW DO OPERATING SYSTEMS REALLY WORK?

Multitasking is not magic! It simplifies and formalizes programming that deals with multiple (usually real-time) activities. Basic switch inputs and display outputs require multitasking just as much as advanced personal computers running jobs for multiple users. The key is in a means to check frequently in the middle of a task for *other* tasks that have become more urgent than the *current* activity.<sup>2</sup>

---

<sup>2</sup>A clever programmer can do that without any additional hardware or software. This becomes very complicated to manage as programs grow larger, and the chance for errors increases when *several* programmers are developing pieces of the overall program.

The pages that follow introduce several commercial multitasking operating systems available for the 8051 family. It would take a whole book to illustrate fully the practical applications of such systems—I will only introduce them.

Keep in mind that any operating system is just additional program code that runs every so often or on request to manage tasks. With multitasking operating systems, *one* high-priority task is the operating system (do not think of it as a task). It comes into play:

1. Any time a running task issues a system call.
2. At each time-increment (system tick—usually an internal timer-caused interrupt), and, depending on the operating system.
3. At any other interrupt.

At such times, the program flow returns to the operating system program. It “wakes up,” recognizes what has happened, and saves the return-address pointer of the program that was running (found on the on-chip stack). It *then* adjusts tables that keep track of task state, interval count, time-out count, etc. Having updated these tables, the operating system program *then* searches through various tables and queues to determine which task to make the running task. Finally, it *switches context*, if required, by *replacing* the return address pointer in the microcomputer’s stack (normally a dangerous practice!) and issuing a *RET* instruction.

From the perspective of the individual tasks you write, a system call is simply a call to a routine, which, after some period of time, “finishes,” and your task’s program flow resumes. The fact that there are *other* tasks is invisible to your task’s program flow.

## Context Switching

Context switching first appeared in Chapter 11 with interrupts. When a particular task is running, it may be that something more urgent will need to have attention of the processor. In any *preemptive* system, rather than having to wait for the first task to finish, it can turn at once to handle the new task and only resume the earlier task when finished. If the program flow instantly switched to the new task, the new task’s re-use of the various registers could destroy intermediate results that the first task was using. When the first task resumed, the changed values in the registers might lead to incorrect results or worse.

The answer to the problem is a set of instructions executed every time a change of task (*context switch*) occurs. Register banks are the hardware

mechanism inherent in 8051 family intended for context switching.<sup>3</sup> One drawback is that only four register banks exist in the 8051 and they only hold 8 bytes each.<sup>4</sup> At most, four different priority levels are possible because you can save only three register bank sets besides the current one.<sup>5</sup>

You can create a system with *more than* four priorities while using only *two* register banks—one for the running task and one for the operating system. For a context switch, the operating system software copies *all* the current registers' contents to off-chip RAM. The newly running task's registers are restored from the same area. The cost is in the longer time to copy the eight registers to off-chip RAM. The operating system must maintain separate, non-overlapping stack areas on-chip.

Where *many* priorities and hundreds of tasks are theoretically desired, some operating systems "clean out" all the on-chip memory. Then the tasks can each have the full on-chip resources—particularly stack space, which is the love of re-entrant C functions. The specific details of the individual systems differ and you will not find the mechanisms clearly described even in the manuals.<sup>6</sup> When you explore purchasing operating systems, be sure to

---

<sup>3</sup>Whenever an interrupt task runs, it can use a different register bank by changing one or two bits in the PSW register. This preserves the 8 bytes (for R0–R7) in the one task and uses a different set of 8 bytes in the interrupt task. At the start of the interrupt, it is advisable to also push the value of the A, B, DPH, and DPL registers onto the stack. This complete context switch thus involves four pushes and a single bit-changing instruction. With on-chip stack, that set of pushes is fast, which was the intent of the 8051 hardware designers from the start.

<sup>4</sup>In the original 8051 design, the intent was to have essentially three priorities. Two priorities of interrupts could preempt the main program. The higher priority would mask off more interrupts, so there could be only two levels of preemption. In that way there are only three register banks needed for context switching. (The 8051 user's manual describes the fourth that comes from a special third-priority scheme.)

<sup>5</sup>Many tasks of *equal* priority are possible, in a round-robin scheme, as long as they do not preempt each other. The DCX operating system used this two-tiered approach.

<sup>6</sup>If a system uses *overlapping* on-chip stack, it probably copies the entire stack for the task to off-chip RAM. If it is going to overlay *other* on-chip variables, it also moves them to a safe place off-chip. In such cases, each task can operate as though virtually all the on-chip resources are at its disposal. The operating system may either blindly copy *everything* or else it cleverly saves only the parts being used by the present task (either by information derived from the compiler/linker files or else by specific inputs from the programmer). You do not need to link together tasks for such systems if the on-chip space can effectively overlap, but the issue of conflict over off-chip RAM may be serious. For the 8051 family that does not have relocatable final code, *that* space can *never* overlap. You should force locating of code and variables to different parts of off-chip RAM for different tasks if they individually locate and link them. I know of no linker that will locate multiple tasks non-overlapping in off-chip RAM and code space while overlaying *all* on-chip resources. Some of the more elaborate operating systems are ported over from other processors with only enough fix-up to run on the 8051—they were designed by "big system" programmers with little appreciation for the limitations of the particular hardware.



ask if the same operating system is available for other processors—usually a bad sign in terms of its fit with the (unusual) architecture of the 8051.

The drawback of the high system approach is speed of context switching. It will take considerable time to move hundreds of bytes to off-chip RAM (with the *MOVX* and *INC DPTR* commands). As a designer, you have to choose whether you need lots of tasks and priorities with lots of variable space on-chip, or whether you need “lean and mean” operation having quick task switching. The answer will probably vary depending on the application and the size of the tasks.<sup>7</sup>

### Setting Priority

In a preemptive system, you can set priorities for tasks.<sup>8</sup> *How to establish priorities is a good question, but start with, How long can this task wait?* For example, take a project that includes the following tasks:

1. Read a character from a serial port before the next character arrives (typically within 1 msec),
2. Recognize a person pushing a button (this could wait for 100 msec before you begin to sense a delay), and
3. Maintain a real-time clock (*ticking*, for example, at a 10 msec rate) by restarting the timer and counting up by one before the next tick expires.

Now to set priorities:

1. Recognize the clock interrupt *at once*, because any delay will slow the clock.<sup>9</sup>
2. The serial input could be *low*-priority, and the button scan could be a non-interrupt *background* task.
3. You might wish instead to make the serial reception *high*-priority because it takes only a few microseconds. With a scheduler you could

---

<sup>7</sup>Personally, it seems inappropriate to use the 8051 for huge applications. Most appropriate applications of the family should run nicely with only a few priorities, but that is admittedly a personal bias. The day may be coming with the 8051 “relatives” where such large applications can fit so nicely that they gain back applications from more advanced processors. Witness the 16-bit “8051s” such as the 251 and the 8051XA.

<sup>8</sup>Even in a nonpreemptive one, you can have higher priority tasks go to the head of the queue although other tasks have been waiting to run for a longer time.

<sup>9</sup>Assuming you must software-reset the timer, with an operating system this clock is automatically the highest priority.

make the real-time clock *low* priority but longer running by including the button scan.

Your decision will depend on the impact of occasionally missing a character or occasionally slipping the clock a few microseconds.<sup>10</sup> Only the specific application can determine the answer to such questions!

## Other Interrupts and Interrupt Handlers

Hardware interrupts are a major problem for the elaborate operating systems, due to context switching. From the perspective of the processor, the operating system is just some code that happens to be running. No special provisions are made. If an interrupt happens, the processor pushes the return address (program counter) onto the stack, masks all interrupts of equal or lower priority, and hands control to the interrupt task at the vector address. The writers of large operating systems are afraid to put such raw power in the programmer's hands, because it is then possible to trash stacks, delay time-slices or time-outs, and generally foul up the system! However, interrupts can get things done quickly without the hassle of a full context switch as long as you do not mess up the registers. If your interrupt routine carefully avoids using them and restores the few it must disturb, it can run quite quickly and the operating system need never know! Consequently, *all* the operating systems reluctantly admit that you can write your own *interrupt handlers*.<sup>11</sup>

For your own *scheduler*, you can use all the hardware interrupts you want as long as they cannot preempt each other.<sup>12</sup> Again, the secret is to

---

<sup>10</sup>If you are writing without a RTOS, because the 8051 directly supports only two levels of interrupt priority along with the background non-interrupt level, limit the use of *different* priorities as much as possible. Most RTOS will mask task interrupts when equal or higher priority tasks are running, so it becomes unavoidable to use different priorities. Put seldom-used tasks that run quickly at a high priority or write them as *interrupt service routines (ISRs)* without context switching at all. One candidate would be a stepper drive task, which uses only a few microseconds every few milliseconds, yet, if it is held up too long, causes the motor to appear to "stutter."

<sup>11</sup>As long as you leave everything masked and do not use the operating system calls within the handler, all is well; you have preempted the *operating system*. If you take too long in your handler, you will block the regular system tick stretch time-outs and time intervals. I assume you will write interrupts carefully to be fast and short—if you need to do lots of processing, set a flag for the main program.

<sup>12</sup>Since there are two priorities of interrupts, if you use different register banks, there is no problem—the hardware blocks equal priority interrupts until the *first* one finishes.

write things as quick-operating as possible and just set flags for processing in the background. That way you will be blind to other interrupts for as little time as possible.

## RESOURCES, REGIONS, POOLS, AND LISTS

Imagine a printer connected to a computer (or a network of computers, if you prefer). Suppose one task/computer is using it to print a several-page document, but, as it pauses to retrieve more data from a file, another task/computer cuts in and begins printing its own job. What a mess that would be! Likewise, if a *data-collecting* task caught up with a *data-processing* task and overwrote some data before it was processed, the old data would be lost and the processing task could be very confused.

Operating systems formalize a protection system for this with an approach using flags or counters to manage what are called *resources*, *regions*, *pools*, or *lists*. A *resource* is usually a hardware item such as a printer's or disk drive. A *region* is a block of memory that is usually "owned" by one task so other tasks cannot mess it up. A *pool* is a shared block of memory that can be broken up and allocated to applications as needed and then returned.<sup>13</sup>

These features differ between systems and are essentially more automatic signaling managed by the operating system to keep one task from accessing variables (or ports or other hardware) before another task is done. They may do the specific function more efficiently, but the equivalent function can be built up from the other system calls and additional programming.

## COMMUNICATION AND SYNCHRONIZATION

Some tasks can function all by themselves, but where tasks work together, they may need to exchange information. This can involve merely a shared variable—a drop-off point—where one task can leave off a pre-arranged value that is a cue to another task to begin some activity. However, you can have a much more formalized exchange with flags, semaphores, and actual messages as events between tasks.<sup>14</sup>

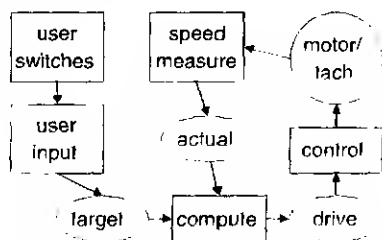
---

<sup>13</sup>I have never actually used the RTOS that manages *lists*, but I believe it can function in controlling access to memory much like a *pool*, or manage resources as well.

<sup>14</sup>These pages illustrate only the use of shared variables and flags. The use of more formal signaling, and message passing techniques are discussed in the companion book.

## Shared Variables for Communication

Consider an example using shared variables for a closed-loop motor speed control system. *One* task gets a desired-speed reading for a motor from a user, and, after converting it to a rotation-period, leaves the new number in a shared location named *target-period*. The *second* task determines the actual time between pulses and leaves an updated value in *actual-period*. The *third* task compares *target-period* with *actual-period* and makes an appropriate adjustment to *drive*. The *fourth* task, a pulse-width motor drive task, uses *drive* to adjust speed.



**Closed-loop—shared variables**

## Drawbacks of Shared Variables

There are drawbacks to communicating with shared variables. In multitasking terms, updating information is not an *event*. The receiving task has no idea when a new piece of information arrives. If the receiving task ought to begin some activity promptly when information arrives or changes, it must frequently check to see if there has been a change. That in itself takes up processor time. In the example above, the tasks would not know directly when the tachometer value or the switch value changed. The computation might have to cycle endlessly regardless of whether or not new data has arrived.

A more fundamental problem from the perspective of multitasking is that there is no inherent protection from loss of information. A task can change a variable *while it is in use by another task*. If a low-priority task is processing a string of variables when a high-priority task modifies the string, the first task could process part new and part old information.<sup>15</sup>

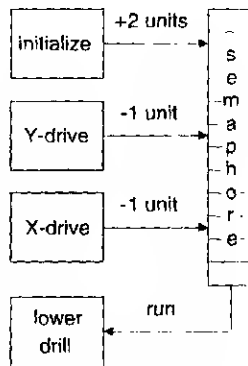
<sup>15</sup>You could set a flag variable to warn the high-priority task to keep out until done, but you will soon have written the equivalent of your own operating system! Unfortunately, this characterizes the evolutionary development of most embedded applications. When the problem has grown so large that a commercial multitasking operating system would be better, you have already invested too much in software development to tolerate a change.

In its simplest form, a binary semaphore is a shared bit variable. A signal could indicate that a task has finished processing a block of memory and an acquisition task is free to refill the block. It can be a simple signal between tasks if the tasks do not need to exchange specific data. Whatever the signal, the various tasks using it must consistently interpret it.

Without an operating system to manage semaphores, you will have to have the waiting task *wake up* every so often to look for bit changes. Otherwise, the waiting task can actually stop permanently. Then you do not need to do any polling because the operating system will cause the task to resume when priorities and flag conditions are right.

### Counting Semaphore

Signaling can involve more than a bit, in which case the semaphore is really a *counter*. The more *units* (count) in the semaphore the greater the *suspension depth* (number of events needed to resume the task's activity).



**Drill-table semaphore**

Consider an example of using a counting semaphore with a printed circuit board drill table. Imagine two tasks (*Xdrive* and *Ydrive*) that need to finish their motions before the *drill-lowering* task can drill the next hole in the printed circuit board. A *startup* task puts *two* units into the drill-lowering task's semaphore and enables the *Xdrive* and *Ydrive* tasks. The drill-lowering task thus begins with a suspension depth of two and waits until there are zero units in its semaphore before running. The *Xdrive* and *Ydrive* tasks each remove *one* unit as they finish their movement, so the semaphore gets to zero only after both motions have finished. You can see how the

term *suspension* arose—the degree to which the task is stopped or *suspended* from running. You avoid having to program to track which of the two tasks finishes first. As you may see, semaphores are quite flexible, but they require imagination to use effectively.

### Messages with Operating System

Most multitasking *operating systems* can also exchange information by passing *messages*. Formal message mechanisms can avoid confusion and save housekeeping.<sup>16</sup> An analogy is letters arriving in a mailbox. Messages, like letters, pile up until you get them out of the mailbox. Here are benefits of operating-system-managed messages over shared variables:

1. Messages are *events*—a task can go to sleep *until a message arrives*. This allows tasks to start up only when the message arrives with no need periodically to use processor time to check the mailbox.
2. You do not have to link tasks using messages together. There is no need to know where the variables are in memory.<sup>17</sup>
3. Messages (usually) queue up for a task, and a later message does not overwrite an earlier one. The task can receive messages in succession. Unlike with shared variables, you cannot overwrite messages before they have been received.

## COMMERCIAL OPERATING SYSTEMS

### DCX51

Intel's *DCX (Distributed Control eXecutive)* was (it is now in the public domain) one example of a multitasking operating system (certainly the first commercial OS for the 8051).<sup>18</sup> It is among the smallest and simplest of them all. There are four pre-defined events for DCX:

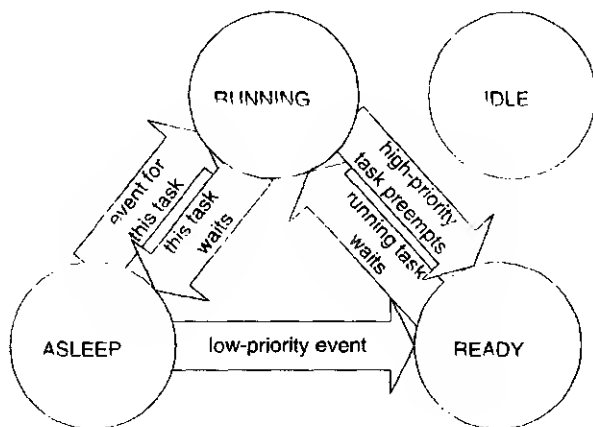
---

<sup>16</sup>With simple, self-written multitasking, it is easiest to share information simply by leaving it off in some agreed-upon array and setting a flag to identify it as new information. The receiving task can clear the flag, indicating that it received the information and the array is free for more new information.

<sup>17</sup>The BITBUS/DCX system was unique in that it actually would send messages over the bus to other processor nodes just as easily as it would send messages between tasks on the same processor.

<sup>18</sup>DCX was the operating system for BITBUS cards. It resided on the 8044 chip (now out of production), but you can still link it in with application code on other 8051 family chips if you abandon the off-chip message features. Both the operating system and a communication task fit within the 4K of on-chip ROM. The system claimed the majority of on-chip RAM for stack and message buffers as well as 300 bytes of off-chip RAM for its various tables and queues. DCX user-tasks had to keep most of their variables in off-chip RAM.

1. *Interrupt* events coming directly from hardware interrupts on the 8051,
2. *Message* events, which are software-initiated by other tasks sending a message to a given task,
3. *Interval* events that are regularly occurring events at some multiple of the system clock (usually 1 msec), and
4. *Time-out* events that come if the other desired events do not take place within a specified time after a task begins waiting.



**DCX task states**

Tasks can be in one of three possible task states. *Running* is the state of a task the processor is executing now. Only one task can be running at a time. *Asleep* is the state of a task that does not need to run until one of the events occurs. *Ready* is the state of a task that *would* run if something of higher priority were not running now. The ready tasks queue up for their turn to run. Finally, there are the *ready/preempted* tasks that *were* running when something of higher priority "woke up" and took over the processor. A preempted task is the first to resume running when its priority level again has use of the processor. This happens when the higher priority task finishes and "goes to sleep."

DCX is virtually nonconfigurable; in making it small, the designers eliminated a number of choices. There can be only eight tasks on record at a time (to make for short tables and queues), one mailbox per task, no semaphores, and only four kinds of events. For context switching, the system uses register banks. This makes context switching quite fast, but limits you to three working priorities (four banks with one taken by the operating system leaves three). With separate internal stacks for each task, there is little space for the normally stack-oriented characteristics of C. Although these

restrictions may bother large-system-oriented programmers. *DCX* adapts well to the constraints of the 8051 family. Now that it is in the public domain, you may be able to locate the source code over the Internet.

### RTX51/RTXTiny

Another operating system, from Keil/Franklin Software, is called *RTX*. It concentrates on the very small applications much like *DCX*. In addition to the *RTXTiny* version, which runs *totally* in on-chip RAM, there is an *RTX51*, which more closely resembles *DCX*. *RTXTiny* uses no more than 64 bytes of RAM depending on how many of the sixteen possible tasks you use. It has code of only about 800 bytes and has only six system calls. *RTX51* is still modest sized, requires some 650 bytes of off-chip RAM, up to 46 bytes of on-chip RAM, and 6 to 8 K of code space. *RTX51* has seventeen system calls, supports nineteen active tasks at one time, and includes message passing as well as timing, interrupts, task signaling, and memory pool management. *RTXTiny* is a “subset” supporting only timing, interrupts, and inter-task signaling, which is enough to build up virtually any application. Both systems will run tasks in round-robin fashion, but the *RTX51* is similar to *DCX* in providing priority levels for tasks. *RTX51* time-slices *equal*-priority tasks whereas *DCX* let equal-priority tasks run to a wait on a first-come/first-served basis.

### Two Basic Groups of RTOS

The two *small* systems, *DCX* and *RTX* differ slightly but they all represent the very small, somewhat restricting approach, which is probably superior for small, fast systems tailored specifically to the 8051. The remaining three systems are quite different in that the developers apparently ported them over to the 8051 from other processors.<sup>19</sup> In preserving some of the features of the operating system on the other processors, efficiency and speed may have suffered. In exchange, you get a range of system calls and a high degree of flexibility that may outweigh the drawbacks. Make your choice guided by the size and complexity of your applications.

### USX

United States Software Corporation markets a multitasking operating system it names *MULTITASK!* It is considerably more complex with thirty-seven system calls. It runs in a round-robin time-slice mode for equal prior-

---

<sup>19</sup>I suspect the developers wrote in C for some other processor and then adapted to the 8051's unusual architecture. That involves some special context switching techniques and, in some cases, assembly language pieces for the most critical parts.



ity tasks, so it can preempt tasks by other equal-priority tasks when the time-slice expires. In a sense, the operating system is a scheduler with interrupt features. It eliminates many limitations of the small systems: There can be 255 tasks with 255 different priorities, a whole host of mailboxes and semaphores as well as resource control. The operating system is a block of C language source, which you compile and link in with your own application code. You supply the C compiler and write your tasks in either C or perhaps assembly. Versions are available for other processors and, as with other products ported over from different processors, the fit with the 8051 is awkward.

USX is highly configurable: It is possible to omit the timing features and get a task sequencer, to omit event and resource controls, or even to omit message capability (no mailboxes). You can limit the size of the various tables by restricting the number of tasks. These configuration choices reduce the size of the system code as well as the RAM usage. The code size is about 10 K for a full system. The literature does not make clear how it manages context switching for the 8051. The system may copy the *entire* on-chip RAM to off-chip space whenever it switches tasks. That would certainly leave a clear place for the next task, but would not be very fast. That could explain why the system favors large tasks and encourages direct user-written interrupt handlers for urgent actions. As long as an interrupt can avoid such a lengthy context switch, you save a lot of time.

## CMX

The CMX Company markets a multitasking operating system that includes a version for the 8051. Like USX, it is a more general software product written in C and ported to the 8051. It has forty-seven system calls and is configurable much like USX. Context switching for the 8051 depends on the particular C compiler used. You may want to check if it supports the particular C compiler you use, since the wide variation in compiler tricks makes it difficult to have a generic operating system while handling non-re-entrant functions.

In addition to message, resource, and event functions, CMX has functions to manage flags (bits within a single byte) and unique ones to manage lists. It makes a distinction between a task and an interrupt. Presumably, you can write an interrupt to quickly modify the appropriate event flags and signal the tasks. Events are strictly task-defined and task-managed. An interrupt handler is the only way to set an interrupt-related event. Probably you would avoid context switching where possible. Both CMX and USX make a

point of specifying a special procedure for calling a system function *from* an interrupt routine.

## Byte-BOS

Byte-BOS Integrated Systems offers an operating system that seems to straddle the large-small gap. It comes in versions specific to the particular C compiler being used (supports at least Keil/Franklin and Whitesmith). Like the previous two RTOS, it is primarily a C program, but it has part of the kernel coded in assembly for the 8051. It too comes in versions for various other processor families.

Byte-BOS combines nonpreemptive and preemptive scheduling. The former allows tasks to run as long as they desire and then schedules the highest priority “ready” task when the running task puts itself to sleep. The task queue is a linked-list so there is no inherent limit on the number of tasks. An interrupt service routine (ISR) invokes the preemptive scheduler. It overrides the currently running task and can “tell” the scheduler to suspend the interrupted task and execute higher priority tasks. Byte-BOS can emulate the simpler preemptive RTOS by adding a user-written ISR to cause rescheduling on every *timer 0* interrupt. The literature indicates that any function that places a task in the ready state will force rescheduling, so the preemptive nature of most system calls is present.

The documentation lists forty-two system calls including ones for managing timers, events (signals), messages, resources, and interrupts. Its complexity is comparable to the other two big systems and the combination of round-robin and preemptive scheduling is fairly complicated at the start. Byte-BOS is unique in having an entire set of calls related to managing of serial communication via a UART.

## Observations on USX, CMX, and Byte-BOS

These three operating systems are significantly larger and more flexible than the first two. The developers did not have just the 8051 in mind and you can have truly ANSI-standard C code to port over to different processor families. Although that may make them more general tools, it *seems* that they are not as carefully fitted to the 8051’s unusual architecture. I would guess they are less efficient although I am sure that various OS developers could argue that. All three have a variety of system calls. You can choose to set up events, resources, and memory pools, but you do have to provide the interface to the specific hardware (including the standard 8051 features relating to interrupts, priorities, and timers). You may get these from sample

files or configuration examples, but you certainly will have more details to handle before starting.

## Benefits of RTOS

1. Real-time operating systems aid in *formalizing* the relationship between activities so you can simplify software development. RTOS are particularly valuable for long projects with *several* programmers where frequent revisions are likely. That probably describes most microcomputer projects! The breaking up of a project into *tasks* allows modular development of the software. In addition to hardware-oriented applications, data processing and reduction can fit a modular, multi-tasking model well.<sup>20</sup> When you break jobs into tasks, it becomes easier to grasp the overall picture without becoming lost in the details. An operating system can make the interrelation between tasks more comprehensible to someone else. With a formal relation between tasks, there will be less chance of overlooking or forgetting details. When the inevitable revisions come, there will be less start-over-from-scratch work.
2. The simplicity of handling *real-time* inputs is a benefit. Multitasking can make it easy to have software respond to real-time demands compared to the single monolithic program approach. In addition, the solutions are more obvious to someone else—not buried in the interrelation of large code modules.
3. *Inter-task communication* (signals and messages) provides a solid method of controlling execution order and timing. In other words, it is possible to move from one activity to another in a way determined by the events that have happened.<sup>21</sup>
4. Tasks can be *small*, making them easier to manage. They can be *independent* so some of the interaction details of single-program solutions are handled by the operating system.

---

<sup>20</sup>For example, one task might be to take a block of raw input data (from an A-D converter) and pick out the high and low values. That in turn might give rise to an alarm message sent to another task that would take corrective action. Additional processing might entail digital signal processing to determine frequency components or to adjust process variables to maintain a desired set point. You could do all this in one block, but it is the *formalizing* of things that helps when several people are working together.

<sup>21</sup>Unlike a scheduler, you do not lock in a task sequence. At one time you can send a message from one task to start a second task running, but at some other time you can send the message to a third task instead.

## Costs of RTOS

What does multitasking *cost*? I have mentioned some of the costs already.

1. You need *hardware*—a specific timer and interrupt structure. Versions of some RTOS exist for other processors but there must be certain hardware in those systems and you must configure the RTOS to match the specific hardware.
2. Multitasking costs *time*. The *overhead* of all the operating system activity uses up computer time. Periodic interrupts to update tables cost time. Sorting through to determine which task should run takes time. If the task relationships are very simple and inflexible, it could be more efficient to write a monolithic program.<sup>22</sup>
3. It takes time for application programmers to get up the *learning curve* far enough to become productive with RTOS. Again, for a single, very small project multitasking probably does not make sense. The formalization and modular nature of the development process can lead to much more *maintainable* code. It is the same problem faced in making a transition from assembly language to a higher level language.

## REVIEW AND BEYOND

1. Why is the handling of an outside interrupt a special problem for some RTOS?
2. In what sort of situations is it important for the receiving task to signal the sending task that it has processed the message?
3. In what sort of applications could you envision a need for a dynamically allocated memory area (pool) shared among tasks?
4. What are the five RTOS described in this chapter? What are the two groupings?
5. Define a semaphore.
6. What are drawbacks in using a shared variable for intertask communication?

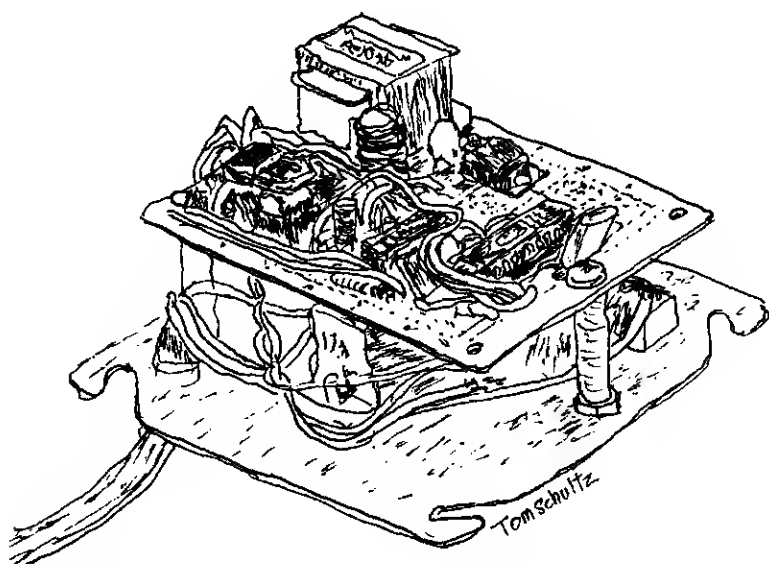
---

<sup>22</sup>If the processing load is too heavy, a different approach may be in order—a faster processor, multiple processors to handle different pieces, or specialized devices such as math coprocessors.

---

# Putting It All Together: An Example

---



This chapter describes both hardware and software of a project illustrating the principles of efficient design. The description is adapted from a report of a project I actually carried out as an example for the students in EET. As you read it, you will notice that the material includes a lot that is not specific to micros. It is my contention that a good project involves extensive planning before the actual hardware and software design starts. The broader the perspective you can bring to the project, the better your choices will be. As I once heard someone say, "If the only tool you have is a hammer, you tend to treat everything as a nail."

The project ended up using a very tiny micro—the 80C751—configured as a scheduler to handle the timing of the frequency measurements for temperature and flow sensors. Notice that I resisted the urge actually to *measure* the two parameters—I just get a relative reading. As I discuss in the companion book, *efficient* design is a different thing from an overly precise design. The essential phrases are “good enough” and “fast enough.”

## CIRCULATING HOT WATER PUMP CONTROLLER

### The Problem: Delayed Hot Water

When someone in a large house with the bathrooms located some distance from a central water heater draws water from a tap until the hot water arrives, the delay is annoying and involves a waste of hot water. I need a more efficient means of making hot water quickly available.

### Background: Hot Water Systems

In the United States, a residential hot water system normally relies on a single heater (typically holding 30 to 40 gal) located in a basement or utility room. This may be at a considerable distance from the actual taps in the kitchen or bathroom. Because of the volume of water in the pipes, it may be some time before the hot water arrives at the tap. This is frustrating and wastes water as well as the energy that previously heated that water.

You could install small under-counter electric water heaters at the distant locations. If the water draw is small or if a large central heater can supply hot water after you deplete the supply in the local tank, this can solve the delay problem. It is, however, expensive and requires space, plumbing, and a source of electricity at the sink. The energy efficiency of the small units depends largely on the amount of insulation on the tanks because it is purely resistive heating.

You could instead plumb a loop so you can bring the hot water around near all the taps and return the cooler water to the heater. Natural convection *can* do this for a short distance, but for long lateral runs, you must add a pump. Such a pump running continuously *does* provide hot water almost instantly. Hotels commonly use it. Unfortunately, the heat loss of a continuously circulating loop, even with well-insulated pipes, contributes a significant increase in energy consumption. In my case the electric bill went up about \$20/month. I next added a timer to the pump to circulate the hot water only at set times (morning and evening,). While this reduced the energy costs, it still has instant hot water available only part of the time.

## THE SOLUTION: AUTOMATIC PUMP CONTROLLER

The chosen approach is to turn on the pump automatically when someone draws water. Using a flow sensor installed near the circulating pump requires no wiring outside the immediate pump/heater area.<sup>1</sup> How can I simply, inexpensively, and reliably determine when someone is drawing water? I could rely on flow or temperature.<sup>2</sup> A flow method gives an immediate and positive indication of water usage (when the pump is off). A temperature method has to rely on either a quick *change* in temperature or a temperature *differential* between the input and the return. I am not sure if the sensing of temperature would be fast enough when you add the several-second delay to get the hot water pumped around the loop once the need is determined.

### Specifications

The proposed pump controller must meet the following specifications:

- Performance:
  - Delivers hot water within 10 s from turn-on of a tap over a distance of at least 100 ft of ¾"-pipe
  - Runs the circulating loop no more than necessary to supply hot water when demanded
  - Requires no special action by the user of hot water
- Installation:
  - Requires access only to the circulating pump area and requires only cutting and gluing of CPVC pipe

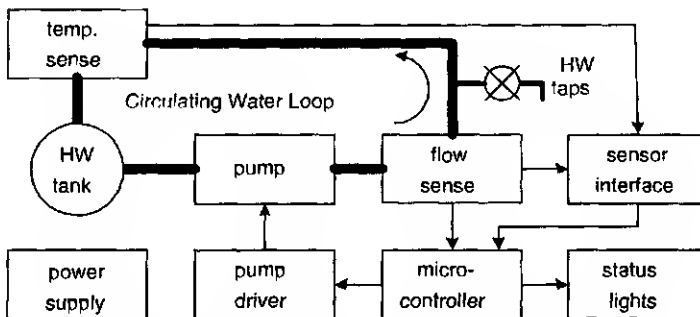
---

<sup>1</sup>As was mentioned, a simple electromechanical timer can be used turn the pump on only at the times of normal usage. This means that users must adjust to the timer schedule or run water for up to 2 min before the hot water arrives at the tap.

A second alternative would be to run low-voltage wiring to each tap and install a button near each tap to push when the user wants hot water. This could signal a one-shot (say a 555 timer IC) to turn on the pump for a fixed time to bring the hot water around. This simple approach requires the installation of wiring in what might be inaccessible places and requires the user to become accustomed to pushing a button for hot water a few seconds before opening the tap. The installation problems would make such an approach unattractive to a general after-market user.

<sup>2</sup>A third option, sensing pressure changes or differentials when water is drawn, was rejected because I feared that the cost would be too high and that pressure fluctuations might also come from other places in the system.

- Requires less than 1 watt of 110 V 50/60 Hz power excluding the power consumed by the pump
- Requires a standard grounded outlet and supplies a similar outlet for the pump
- Special skills beyond those for assembling CPVC plumbing and mounting an electrical box are not required
- Unit self-calibrating with warning lights when not functioning properly
- Physical:
  - Total unit less than 4" × 4" × 4" with bracket to mount to studs or floor joist
  - Weighs less than 10 lb
  - Unharmd by a drop from 8 ft to concrete floor
- Environment:
  - Operates over 0 to 70°C, 0 to 95% humidity
  - Minimum moving parts, function reliably for 10 years with hard or iron-rich water as well as soft water
  - Operates with water up to 180°F and 100 psi
- Cost:
  - Prototype parts cost less than \$100
  - Production units (including sensors) parts cost less than \$50 in 1000 quantity
  - Electric bill in prototype installation should drop by at least \$10 per month



**Overall block diagram of hot water controller**



## User's Instructions

You do not have to do anything special to use the installed system. The only change is that the hot water is available almost at once whenever you open the hot water tap.

The installation assumes you have a hot water loop and pump. Retrofitting a hot water loop is only practical in houses where the most distant hot tap is accessible and you can fit a return pipe without tearing out walls. You can install a pump (available for about \$100) at the same time as the sensors.

To install, shut off the water and drain the pipes. Be sure to shut off the water heater if you must drain it.

Mount the controller box to a joist or stud so that the cord reaches a standard 110 V outlet and so that the controller outlet is within reach of the circulating pump power cord (it may be necessary to attach a three-prong cord to the pump). The ON-OFF-AUTO switch and light should be visible to the homeowner.

Cut the outgoing (CPVC) hot water pipe at a vertical run near the pump and remove a 6"-length of pipe. Glue in the supplied flow sensor making sure the side with the coil is down. (Copper plumbing will need the alternate kit with adapters.) Connect the sensor wires from the pipe assembly to the controller terminals labeled "flow." Polarity does not matter.

Cut the incoming (return) pipe somewhere after the last tap in the loop and remove a 3"-length of pipe. Glue in the supplied temperature sensor and run the sensor wires to the controller terminals. Observe polarity on these connections.

Switch the controller to *OFF*, turn on the water and check to be sure there are no leaks. Then plug in the controller and plug the pump into the controller. Switch the unit to *ON* and listen to be sure the pump is running. Shut it off again and allow the system to settle to a cold state (no hot water drawn for at least 20 min). Then switch to *AUTO* and observe the unit self-calibrating. The green light should flash and the pump should run for about 2 min. At the end of that time, the green light should remain on steadily. Any faults will cause the red light to come on.

## PRINCIPLES OF OPERATION

### Flow Sensor

The flow is sensed using a coil wrapped around a length of plastic pipe with a ball bearing trapped inside. When the flow starts, the ball moves up out of the coil, changing the inductance (and the resulting frequency of an LC oscillator). You sense that change of inductance (frequency) with the microcontroller. The frequency shift is not critical since the microcontroller will determine the no-flow and full-flow values at calibration time (it controls the pump!). You can easily resolve a change of a few tens of hertz out of a few kHz. The capacitor choice and the inductance of the coil determine the overall frequency.<sup>3</sup> I tested the flow sensor and interface together by moving the ball bearing inside the pipe and observing the frequency shift.

### Temperature Sensor

Temperature sensing is not vital to the project. It is included as a supplementary device for the calibration phase—run the loop until the return is hot to determine how long it takes to flush the loop of cold water. If there were no temperature sensor, the pump could easily run for a *fixed* time with no significant loss of system performance.

As mentioned earlier, I rejected relying on temperature sensing to detect drawing of hot water because of the expected delay. Sensing temperature is done with a thermistor inserted in the pipe.<sup>4</sup> Since the flow system al-

---

<sup>3</sup> Alternate solutions considered included an internal paddle or hinged flap. The paddle would be good for measuring the actual flow rate, but it is much too complex and expensive for the cost goals here and sediment or hardness buildup might affect it. The paddle pick-up could work given a magnet attached to one part with a reed switch to sense each turn. The goal would be to avoid any holes in the pipe that could leak with a moving shaft.

The flap idea could also work with a magnet to sense when the flap has moved away from vertical (must be in a horizontal length of pipe with the hinge at the top). The entire assembly of such a system is much more complicated than a simple ball bearing and more critical for installation.

Other noncontact flow-sensing methods such as ultrasonic Doppler or optical methods are much too complex and expensive here, where the specific rate of flow is of no concern.

<sup>4</sup> To keep it from obstructing the flow, the thermistor is mounted inside the pipe in the side branch of a "T" with sealed leads coming out of two small holes in a plastic pipe cap glued to the stub.

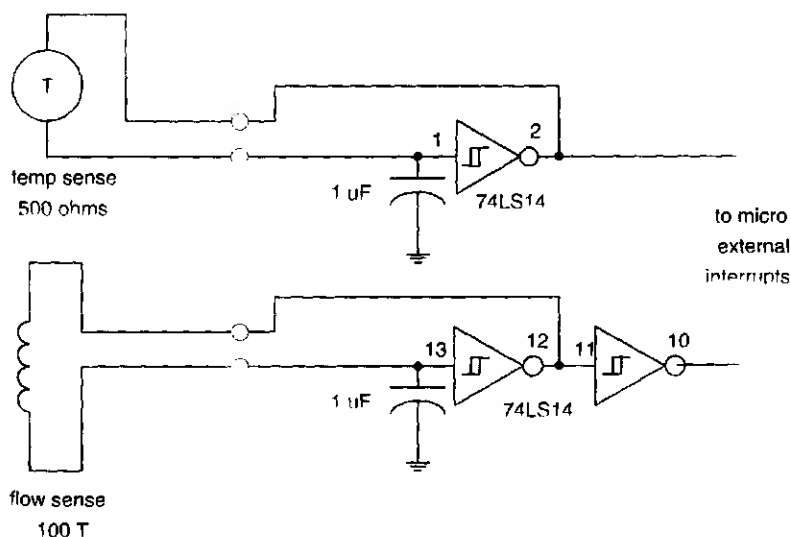
One alternate temperature sense approach is the use of an LM34 solid-state sensor, but it was rejected because of thermal mass and cost. It also does not adapt to being an oscillator element as readily as a thermistor—its resistance does not change as much over the temperature range of interest. Linearity and accuracy are of no concern here anyway.

Another alternative, thermocouples, are much more complicated to use because of the low-level signals. There are, I understand, temperature sensors available now that include the frequency conversion with the sensor, but I think the cost would be higher when only a *relative* temperature measurement is needed.

ready uses an oscillator, the thermistor is put in an RC oscillator circuit and the reading is also brought in as a frequency. Again, I do not need *absolute* temperature since I only sense a *change* during the calibration phase. The thermistor's nonlinearity is of no concern here as long as there is an adequate frequency shift between "cold" and "hot." External sensing was explored with a length of copper pipe and an attached thermistor, but the thermal time-constant ruled it out and splicing copper into a plastic system is significantly more expensive. The temperature and interface blocks are tested together by immersing the thermistor in hot and cold water alternately and observing the frequency shift and response delay.

### Sensor Interface

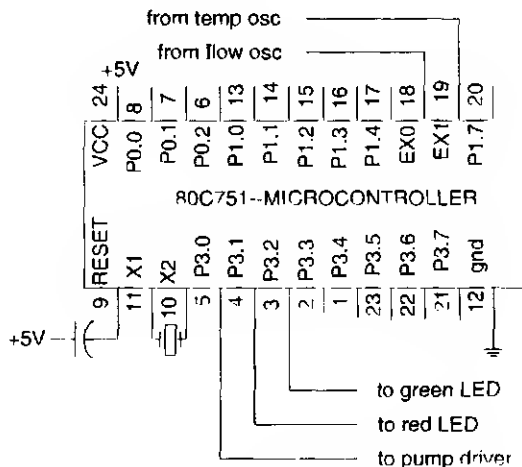
I have already described the sensors. The interface to the microcontroller consists of a pair of oscillators made with Schmidt triggers. The 7414 is inexpensive and directly TTL-compatible for the microcontroller. Frequency stability and repeatability from chip to chip are of no concern because the microcontroller self-calibrates the system. The temperature interface has no output huffer because there were no gates left in the (hex) package and that circuit had a better wave shape without the huffer than the flow circuit.



**Sensor and interface oscillator schematics**

## Microcontroller

This is a single-chip 87C751, which is a thin twenty-four-pin member of the 8051 family. It easily provides the five I/O lines needed. It has two internal timers that provide a real-time clock and a reference for frequency measurement from the two sensors.<sup>5</sup> The drive to the triac/pump is a single-output bit, and a pair of bits drives the status LEDs.



**Microcontroller I/O diagram**

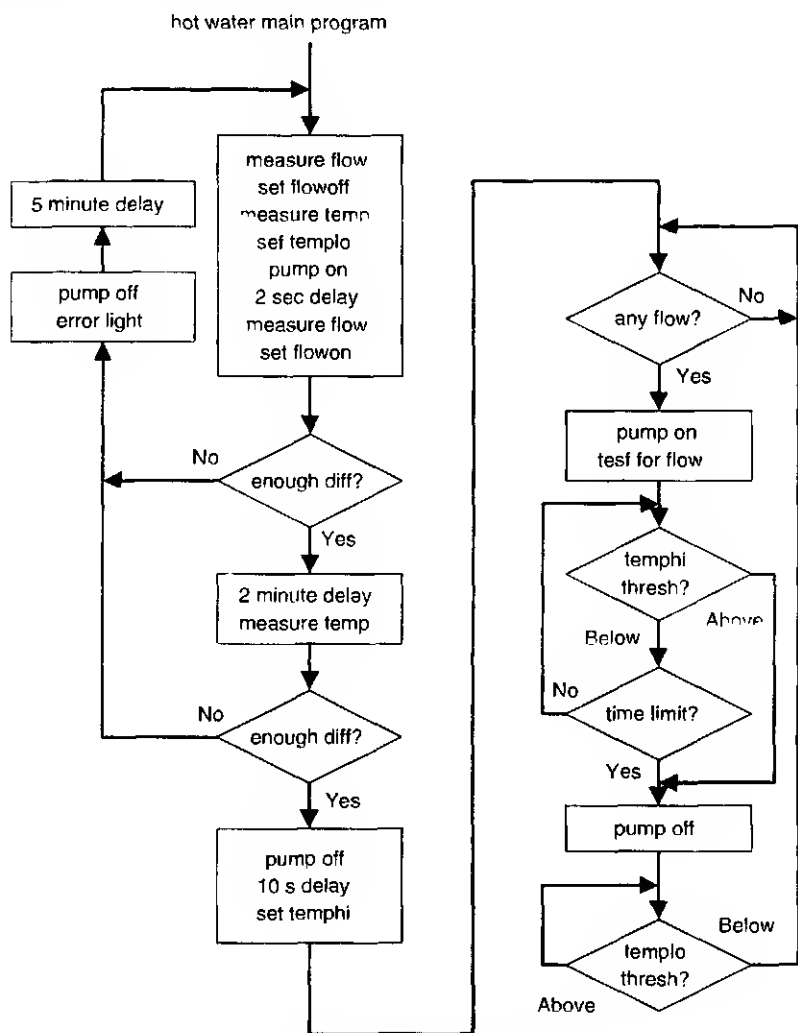
A complete commercial microcontroller board is inconsistent with the overall size and cost requirements. Other 8051 family single-chip devices could work, but they are significantly larger and are overkill for the simple requirements.<sup>6</sup> The microcontroller hardware was first tested with a simple program burned into the internal (erasable version) EPROM. The test program toggled one of the port pins at a 1 kHz rate.

<sup>5</sup>I could have used the 750 except it only has a single timer. (It does not have the fixed-rate timer that is usually involved with the I<sup>2</sup>C bus interface.)

<sup>6</sup>I rejected the 4-bit processor families because they have fewer software development resources and, I believe, do not have built-in timer functions. A program using a software timing loop might be possible with these, but I chose to avoid an entire new processor at this time. Other 8-bit microcontroller families were also not considered for a book on the 8051! If mass production appeared likely with correspondingly severe cost constraints, then someone could reopen this issue.

While a discrete logic design might have met most of the project requirements, the lack of self-calibration and the expected higher chip count worked against the size requirements and would cost more in production.

The software is extremely simple if you overlook self-calibration—if *flow is sensed, turn on the pump for 1 min and then ignore flow for at least 15 min* (since the water in the loop remains hot enough for about that long). As it is, with the frequency inputs and the timer for general clock functions, the software is still a reasonable size. Following good programming techniques, all the software is written in modular form with several separate functions. By separating the time and frequency functions from the main program, a kind of multitasking system has emerged.



**Overall hotwater flowchart**

**Main Hotwater Function**


---

```

#define uchar unsigned char
#define uint unsigned
#define forever for(;;)
#define run 1
#define stop 0
#define redgrnon 0x0f
#define grnsteady 0x0a
#define redsteady 0x05
#define grnslow 0x02
#define grnfast 0x82
#define redslow 0x01
#define redfast 0x81
#define tempdiff 5
#define timelimit 8
sfr TH=0x8a;
sfr TL=0x8c;
sfr RTL=0x8b;
sfr RTH=0x8d;
sfr IE=0xa8;
sfr TCON=0x88;
sbit redled=0xb2;
sbit grnled=0xb1;
sbit pump=0xb0;
uchar flow,on,off;
uchar temp,hot,cold,x,j,i,time;
uchar trdg[8];
uint msecs;
uchar LEDS;

/*****/
/* this watches the msec variable set by */
/* timer1 interrupt.  it burns time badly*/
/* but nothing else needs doing          */
/*****/
void delay(uint x){
    uint endtime=msecs+x;
    while(msecs!=endtime);
}

void main(void){
    RTH=0; RTL=0; /*full count*/
    TCON=0x15; /*timer en., exints edge trig.*/
    IE=0x8d; /*exts and timer*/

```

```

delay(1000); /*to get first readings*/
LEDS=redgrnon;
do{
    do{
        off=flow;
        cold=temp;
        pump=run;
        LEDS=grnslow;
        delay(2000);
        on=flow;
        LEDS=redslow;
    }while((on-off)<8);
    for(j=0;j<8;j++){
        trdg[j]=temp;
        delay(15000);
    }
}while((x=(trdg[0]-trdg[7]))<tempdiff);
for(j=0;trdg[j]-trdg[0]<8*x/10||j<8;j++);
    /*j=# of 15 sec times for hot-2 min max */
pump=stop;
LEDS=grnsteady;
forever{
    while(flow<on); /*wait for use of water*/
    pump=run;
    LEDS=grnslow;
    time=0;
    while ((temp<hot)&&(time<timelimit)){
        delay(10000); time++;
    }
    pump=stop;
    if(time==timelimit) LEDS=redfast;
    else LEDS=grnsteady;
    for(i=0;i<10;i++)delay(60000);
                                   /*10 minute cool down*/
}
}

```

---

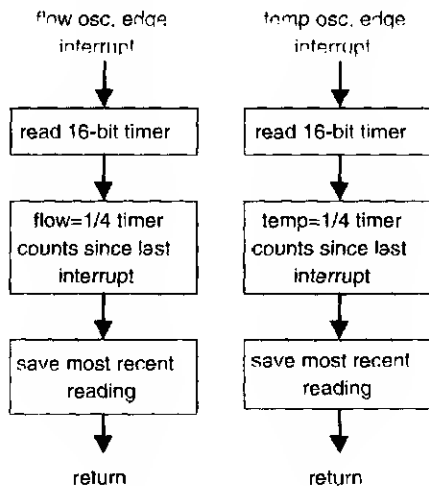
## Multitasking Features

The most interesting microcontroller feature of this project, from my perspective, is the way I can measure two frequencies with a single timer

using interrupt-driven routines. A first approach to measuring frequency might be to count incoming edges for a fixed time. The higher the frequency, the more counts—1000/s, for example. However, that ties up a counter to count the edges and a timer to clock the interval. If there are two frequencies, as in this case, that takes *two* counters *and* a timer. I could do that with an 80C52, for example, but definitely not with the 80C750 or 80C751.

The solution is to time *one cycle* of the frequency. The lower the frequency, the longer the period. If you have control of the V-F circuit design, make the frequency very low by choosing large capacitors. Then a cycle will give a higher count for better resolution. If that resolution is insufficient, count for several or even hundreds of cycles.

The key to accomplishing all this with one timer is to either poll or use the external interrupts. I took the latter approach in this case. The two external interrupt routines (flow and temperature) are identical except for the interrupt number. When an interrupt comes—the negative *edge* of the frequency input—the interrupt routine captures the count in the free-running 16-bit timer and subtracts it from the count at the time of the previous edge. That is the time for the one cycle. The timer free-runs, the interrupts go off the edges of the frequencies, and everything else goes on quite independently. The interrupt routines update the shared flow and temp variables each time so the main program just assumes the value is always current.



**Flow and temperature measurement flowcharts**



---

### Temperature and Flow Measurement Functions

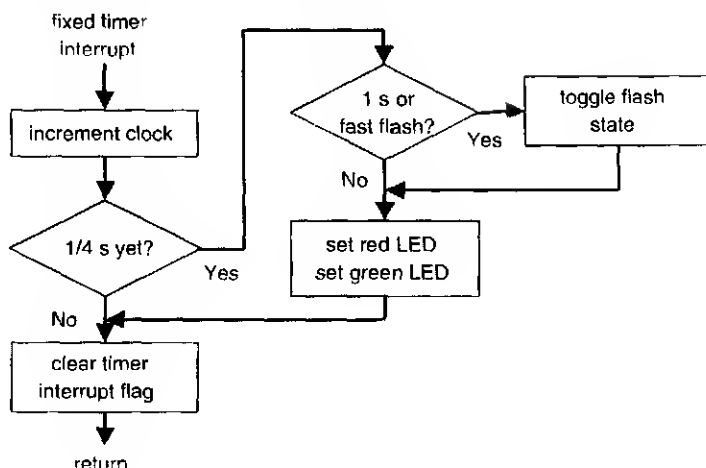
```
#define uchar unsigned char
sfr TH=0x8a;
sfr TL=0x8c;
extern uchar flow, temp; /*global variables*/
/*****
/* uses v-f edges to get time difference */
/* representing the frequency due to the */
/* ball in or out of the coil */
*****/
void flowint(void) interrupt 1 using 1{
    uchar fnew;
    static uchar fold;
    fnew=((uint) TL) | (((uint) TH)<<8);
    flow=(fnew-fold)>>2;
    fold=fnew;
}
/*****
/* uses v-f edges to get time difference */
/* representing the frequency due to the */
/* resistance of the thermistor */
*****/
void tempint(void) interrupt 3 using 1{
    uchar tnew;
    static uchar told;
    tnew=((uint) TL) | (((uint) TH)<<8);
    temp=(tnew-told)>>2;
    told=tnew;
}
```

---

I could do the timing for the delays with the free-running 16-bit timer,<sup>7</sup> but I chose to use the fixed rate timer that would otherwise run the I<sup>2</sup>C bus (not present in the 750). It takes care of updating a time variable for delays and acts as a *scheduler* for the flashing of the LEDs by counting the timer interrupts. Therefore, without having an operating system, using the smallest 8051 member, you get a multitasking system!

---

<sup>7</sup>The big drawback in leaving the 16-bit timer free running so the subtraction doesn't require special treatment is in producing a time increment out of 65,536 counts. At 12 MHz (1 MHz count rate), that is about 65 msec, or 15.2587 cycles/s—not a nice round number. Now 915.522 cycles would be a minute, if that was the smallest time increment you needed, or 54931.32 cycles would be an hour! The point is that precise, round-number timing isn't as simple with the free-running counter.



Clock and LED flash flowchart

---

### Clock and LED Flashing Functions

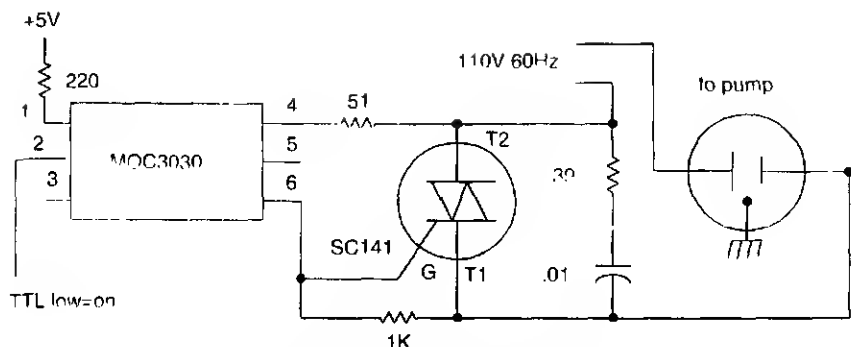
```

#define uchar unsigned char
sbit IT1=0x88;
sbit redled=0xb2;
sbit grnled=0xb1;
extern uchar LEDS;
extern uint msec;
/*****
/* uses the regular 1 msec interrupt to */
/* keep a count of time (msec) and also */
/* handles the flashing of the leds. */
/* LEDS codes are in the defines (at top)*/
*****/
void timer1 (void) interrupt 4 using 1{
    bit flashstate;
    static uchar ctr;
    uchar temp,j;
    if(++msec&0xff==0){ /*abt. every 1/4 sec*/
        if((++ctr%3)==0){ (LEDS&0x80)
flashstate=~flashstate;
        if(flashstate)temp=LEDS&3;
        else temp=(LEDS>>2)&3;
        redled=temp&1;
        grnled=temp>>1&1;
        }
    }
    IT1=0;
}

```

---

## Pump Drive

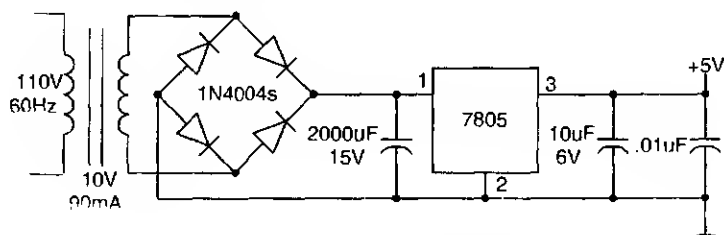


**Pump drive schematic**

The pump requires only about 1/2 amp of 110 V, so it can be driven by a simple 5 V relay or a triac. The triac is more expensive if you include an isolated driver (MOC3030), but it should have a longer life.<sup>8</sup> I first tested the pump drive circuit by turning on and off a 50 W light bulb, grounding and opening the input to the opto-isolator driving the triac. Then I drove it with a simple program from the micro turning it on and off at a 1 s rate. Only in the integration phase did I use an actual pump.

## Power Supply

The entire control unit operates off 5 V and draws less than 100 mA. In keeping with the size requirements, I used a small transformer to make a linear supply with a three-terminal regulator. I then only needed a small electrolytic capacitor and a few 1N4004 diodes. I first tested the power supply with a resistive load of 50 Ω to check for ripple and heating problems as well as to be sure of the accuracy of regulation.

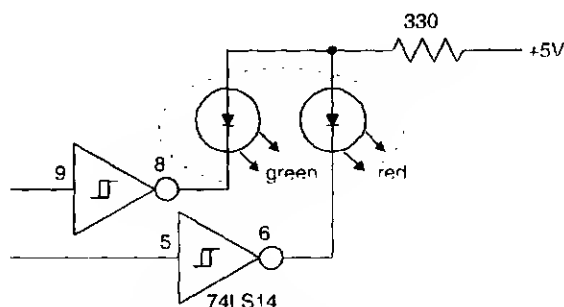


**Power supply schematic**

<sup>8</sup>For safety reasons, I decided to isolate rather than to float the entire unit and drive the triac directly—particularly because the thermistor goes directly into the water where isolation of the 110 V could be difficult and risky.

## Status Indicator

A single “two-color” LED reduces the cost of mounting hardware. Otherwise, two diodes might be less expensive.<sup>9</sup> Having the microcontroller sequence through the four possible states on the two port lines and observing the red and green colors tested the LED and driver.



**LED status indicator schematic**

## Mode Switch

To avoid frustration if there is a failure, I provide a manual override switch. It bypasses the microcontroller completely to shut the pump off or leave it always on. The *OFF* setting cuts the power to the microcontroller losing the previous calibration settings. The AC power wiring is here in the overall schematic.

---

<sup>9</sup>I rejected LCD displays or seven-segment LEDs because of cost as well as the fact that the user or the installer needs very little information.



# SECTION IV

## Appendices

---

Appendix A1 consists of two tables listing the machine instructions first in numeric order by the machine codes and then in alphabetic order by the assembly mnemonics.

Appendix A2 repeats the development steps of Chapter 9, but uses only DOS rather than the windowed environment.

Appendix A3 gives hints for switching from some other language to the C or assembly of the 8051. It is particularly useful for anyone switching from PL/M or from more conventional C.

Appendix A4 describes bank switching and gives some detail about two development boards—the PU552 and the MCB520.

Appendix A5 describes the different members of the 8051 family along with a brief description of some of the added features.

Appendix A6 is a collection of addresses and telephone numbers for various hardware and software suppliers.



# A1

## Instructions: Numeric and Alphabetic Order

---

### NUMERIC ORDER

Hex Code	Bytes	Mnemonic	Operand
0	1	NOP	
1	2	AJMP	codeaddr
2	3	LJMP	codeaddr
3	1	RR	A
4	1	INC	A
5	2	INC	dataaddr
6	1	INC	@R0
7	1	INC	@R1
8	1	INC	R0
9	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bitaddr, codeaddr
11	2	ACALL	codeaddr
12	3	LCALL	codeaddr
13	1	RRC	A
14	1	DEC	A

*(continued)*



Hex Code	Bytes	Mnemonic	Operand
15	2	DEC	dataaddr
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	bitaddr, codeaddr
21	2	AJMP	codeaddr
22	1	RET	
23	1	RL	A
24	2	ADD	A, #data
25	2	ADD	A, dataaddr
26	1	ADD	A, @R0
27	1	ADD	A, @R1
28	1	ADD	A, R0
29	1	ADD	A, R1
2A	1	ADD	A, R2
2B	1	ADD	A, R3
2C	1	ADD	A, R4
2D	1	ADD	A, R5
2E	1	ADD	A, R6
2F	1	ADD	A, R7
30	3	JNB	bitaddr, codeaddr
31	2	ACALL	codeaddr
32	1	RET	
33	1	RLC	A
34	2	ADDC	A, #data
35	2	ADDC	A, dataaddr
36	1	ADDC	A, @R0
37	1	ADDC	A, @R1
38	1	ADDC	A, R0
39	1	ADDC	A, R1
3A	1	ADDC	A, R2
3B	1	ADDC	A, R3

(continued)

Hex Code	Bytes	Mnemonic	Operand
3C	1	ADDC	A, R4
3D	1	ADDC	A, R5
3E	1	ADDC	A, R6
3F	1	ADDC	A, R7
40	2	JC	codeaddr
41	2	AJMP	codeaddr
42	2	ORL	dataaddr, A
43	3	ORL	dataaddr, #data
44	2	ORL	A, #data
45	2	ORL	A, dataaddr
46	1	ORL	A, @R0
47	1	ORL	A, @R1
48	1	ORL	A, R0
49	1	ORL	A, R1
4A	1	ORL	A, R2
4B	1	ORL	A, R3
4C	1	ORL	A, R4
4D	1	ORL	A, R5
4E	1	ORL	A, R6
4F	1	ORL	A, R7
50	2	JNC	codeaddr
51	2	ACALL	codeaddr
52	2	ANL	dataaddr, A
53	3	ANL	dataaddr, #data
54	2	ANL	A, #data
55	2	ANL	A, dataaddr
56	1	ANL	A, @R0
57	1	ANL	A, @R1
58	1	ANL	A, R0
59	1	ANL	A, R1
5A	1	ANL	A, R2
5B	1	ANL	A, R3
5C	1	ANL	A, R4
5D	1	ANL	A, R5
5E	1	ANL	A, R6
5F	1	ANL	A, R7
60	2	JZ	codeaddr
61	2	AJMP	codeaddr
62	2	XRI	dataaddr, A

(continue)

Hex Code	Bytes	Mnemonic	Operand
63	3	XRL	dataaddr, #data
64	2	XRL	A, #data
65	2	XRL	A, dataaddr
66	1	XRL	A, @R0
67	1	XRL	A, @R1
68	1	XRL	A, R0
69	1	XRL	A, R1
6A	1	XRL	A, R2
6B	1	XRL	A, R3
6C	1	XRL	A, R4
6D	1	XRL	A, R5
6E	1	XRL	A, R6
6F	1	XRL	A, R7
70	2	JNZ	codeaddr
71	2	ACALL	codeaddr
72	2	ORL	C, bitaddr
73	1	JMP	@A+DPTR
74	2	MOV	A, #data
75	3	MOV	dataaddr, #data
76	2	MOV	@R0, #data
77	2	MOV	@R1, #data
78	2	MOV	R0, #data
79	2	MOV	R1, #data
7A	2	MOV	R2, #data
7B	2	MOV	R3, #data
7C	2	MOV	R4, #data
7D	2	MOV	R5, #data
7E	2	MOV	R6, #data
7F	2	MOV	R7, #data
80	2	SJMP	codeaddr
81	2	AJMP	codeaddr
82	2	ANL	C, bitaddr
83	1	MOVC	A, @A+PC
84	1	DIV	AB
85	3	MOV	dataaddr, dataaddr
86	2	MOV	dataaddr, @R0
87	2	MOV	dataaddr, @R1
88	2	MOV	dataaddr, R0
89	2	MOV	dataaddr, R1

*(continued)*

Hex Code	Bytes	Mnemonic	Operand
8A	2	MOV	dataaddr, R2
8B	2	MOV	dataaddr, R3
8C	2	MOV	dataaddr, R4
8D	2	MOV	dataaddr, R5
8E	2	MOV	dataaddr, R6
8F	2	MOV	dataaddr, R7
90	3	MOV	DPTR, #data
91	2	ACALL	codeaddr
92	2	MOV	bitaddr, C
93	1	MOVC	A, @A+DPTR
94	2	SUBB	A, #data
95	2	SUBB	A, dataaddr
96	1	SUBB	A, @R0
97	1	SUBB	A, @R1
98	1	SUBB	A, R0
99	1	SUBB	A, R1
9A	1	SUBB	A, R2
9B	1	SUBB	A, R3
9C	1	SUBB	A, R4
9D	1	SUBB	A, R5
9E	1	SUBB	A, R6
9F	1	SUBB	A, R7
A0	2	ORL	C, /bitaddr
A1	2	AJMP	codeaddr
A2	2	MOV	C, bitaddr
A3	1	INC	DPTR
A4	1	MUL	AB
A5			reserved
A6	2	MOV	@R0, dataaddr
A7	2	MOV	@R1, dataaddr
A8	2	MOV	R0, dataaddr
A9	2	MOV	R1, dataaddr
AA	2	MOV	R2, dataaddr
AB	2	MOV	R3, dataaddr
AC	2	MOV	R4, dataaddr
AD	2	MOV	R5, dataaddr
AE	2	MOV	R6, dataaddr
AF	2	MOV	R7, dataaddr
B0	2	ANL	C, /bitaddr

(continued)

Hex Code	Bytes	Mnemonic	Operand
B1	2	ACALL	codeaddr
B2	2	CPL	bitaddr
B3	1	CPL	C
B4	3	CJNE	A, #data, codeaddr
B5	3	CJNE	A, dataad, codead
B6	3	CJNE	@R0, #data, codead
B7	3	CJNE	@R1, #data, codead
B8	3	CJNE	R0, #data, codeadd
B9	3	CJNE	R1, #data, codeadd
BA	3	CJNE	R2, #data, codeadd
BB	3	CJNE	R3, #data, codeadd
BC	3	CJNE	R4, #data, codeadd
BD	3	CJNE	R5, #data, codeadd
BE	3	CJNE	R6, #data, codeadd
BF	3	CJNE	R7, #data, codeadd
C0	2	PUSH	dataaddr
C1	2	AJMP	codeaddr
C2	2	CLR	bitaddr
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A, dataaddr
C6	1	XCH	A, @R0
C7	1	XCH	A, @R1
C8	1	XCH	A, R0
C9	1	XCH	A, R1
CA	1	XCH	A, R2
CB	1	XCH	A, R3
CC	1	XCH	A, R4
CD	1	XCH	A, R5
CE	1	XCH	A, R6
CF	1	XCH	A, R7
D0	2	POP	dataaddr
D1	2	ACALL	codeaddr
D2	2	SETB	bitaddr
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	dataaddr, codeaddr
D6	1	XCHD	A, @R0
D7	1	XCHD	A, @R1

(continued)

Hex Code	Bytes	Mnemonic	Operand
D8	2	DJNZ	R0, codeaddr
D9	2	DJNZ	R1, codeaddr
DA	2	DJNZ	R2, codeaddr
DB	2	DJNZ	R3, codeaddr
DC	2	DJNZ	R4, codeaddr
DD	2	DJNZ	R5, codeaddr
DE	2	DJNZ	R6, codeaddr
DF	2	DJNZ	R7, codeaddr
E0	1	MOVX	A, @DPTR,
E1	2	AJMP	codeaddr
E2	1	MOVX	A, @R0
E3	1	MOVX	A, @R1
E4	1	CLR	A
E5	2	MOV	A, dataaddr
E6	1	MOV	A, @R0
E7	1	MOV	A, @R1
E8	1	MOV	A, R0
E9	1	MOV	A, R1
EA	1	MOV	A, R2
EB	1	MOV	A, R3
EC	1	MOV	A, R4
ED	1	MOV	A, R5
EE	1	MOV	A, R6
EF	1	MOV	A, R7
F0	1	MOVX	@DPTR, A
F1	2	ACALL	codeaddr
F2	1	MOVX	@R0, A
F3	1	MOVX	@R1, A
F4	1	CPL	A
F5	2	MOV	dataaddr, A
F6	1	MOV	@R0, A
F7	1	MOV	@R1, A
F8	1	MOV	R0, A
F9	1	MOV	R1, A
FA	1	MOV	R2, A
FB	1	MOV	R3, A
FC	1	MOV	R4, A
FD	1	MOV	R5, A
FE	1	MOV	R6, A
FF	1	MOV	R7, A

## INSTRUCTIONS SORTED ALPHABETICALLY

Mnemonic	Operand	Bytes	Hex Code
—	reserved		A5
ACALL	codeaddr	2	11
ACALL	codeaddr	2	31
ACALL	codeaddr	2	51
ACALL	codeaddr	2	71
ACALL	codeaddr	2	91
ACALL	codeaddr	2	B1
ACALL	codeaddr	2	D1
ACALL	codeaddr	2	F1
ADD	A, #data	2	24
ADD	A, @R0	1	26
ADD	A, @R1	1	27
ADD	A, dataaddr	2	25
ADD	A, R0	1	28
ADD	A, R1	1	29
ADD	A, R2	1	2A
ADD	A, R3	1	2B
ADD	A, R4	1	2C
ADD	A, R5	1	2D
ADD	A, R6	1	2E
ADD	A, R7	1	2F
ADDC	A, #data	2	34
ADDC	A, @R0	1	36
ADDC	A, @R1	1	37
ADDC	A, dataaddr	2	35
ADDC	A, R0	1	38
ADDC	A, R1	1	39
ADDC	A, R2	1	3A
ADDC	A, R3	1	3B
ADDC	A, R4	1	3C
ADDC	A, R5	1	3D
ADDC	A, R6	1	3E
ADDC	A, R7	1	3F
AJMP	codeaddr	2	1
AJMP	codeaddr	2	21
AJMP	codeaddr	2	41
AJMP	codeaddr	2	61

(continued)

Mnemonic	Operand	Bytes	Hex Code
AJMP	codeaddr	2	81
AJMP	codeaddr	2	A1
AJMP	codeaddr	2	C1
AJMP	codeaddr	2	E1
ANL	A, #data	2	54
ANL	A, @R0	1	56
ANL	A, @R1	1	57
ANL	A, dataaddr	2	55
ANL	A, R0	1	58
ANL	A, R1	1	59
ANL	A, R2	1	5A
ANL	A, R3	1	5B
ANL	A, R4	1	5C
ANL	A, R5	1	5D
ANL	A, R6	1	5E
ANL	A, R7	1	5F
ANL	C, /bitaddr	2	B0
ANL	C, bitaddr	2	82
ANL	dataaddr, #data	3	53
ANL	dataaddr, A	2	52
CJNE	@R0, #data, codead	3	B6
CJNE	@R1, #data, codead	3	B7
CJNE	A, #data, codeaddr	3	B4
CJNE	A, dataad, codead	3	B5
CJNE	R0, #data, codeadd	3	B8
CJNE	R1, #data, codeadd	3	B9
CJNE	R2, #data, codeadd	3	BA
CJNE	R3, #data, codeadd	3	BB
CJNE	R4, #data, codeadd	3	BC
CJNE	R5, #data, codeadd	3	BD
CJNE	R6, #data, codeadd	3	BE
CJNE	R7, #data, codeadd	3	BF
CLR	A	1	E4
CLR	bitaddr	2	C2
CLR	C	1	C3
CPL	A	1	F4
CPL	bitaddr	2	B2
CPL	C	1	B3
DA	A	1	D4

(continued)



Mnemonic	Operand	Bytes	Hex Code
DEC	@R0	1	16
DEC	@R1	1	17
DEC	A	1	14
DEC	dataaddr	2	15
DEC	R0	1	18
DEC	R1	1	19
DEC	R2	1	1A
DEC	R3	1	1B
DEC	R4	1	1C
DEC	R5	1	1D
DEC	R6	1	1E
DEC	R7	1	1F
DIV	AB	1	84
DJNZ	dataaddr, codeaddr	3	D5
DJNZ	R0, codeaddr	2	D8
DJNZ	R1, codeaddr	2	D9
DJNZ	R2, codeaddr	2	DA
DJNZ	R3, codeaddr	2	DB
DJNZ	R4, codeaddr	2	DC
DJNZ	R5, codeaddr	2	DD
DJNZ	R6, codeaddr	2	DE
DJNZ	R7, codeaddr	2	DF
INC	@R0	1	6
INC	@R1	1	7
INC	A	1	4
INC	dataaddr	2	5
INC	DPTR	1	A3
INC	R0	1	8
INC	R1	1	9
INC	R2	1	0A
INC	R3	1	0B
INC	R4	1	0C
INC	R5	1	0D
INC	R6	1	0E
INC	R7	1	0F
JB	bitaddr, codeaddr	3	20
JBC	bitaddr, codeaddr	3	10
JC	codeaddr	2	40
JMP	@A+DPTR	1	73

(continued)

Mnemonic	Operand	Bytes	Hex Code
JNB	bitaddr, codeaddr	3	30
JNC	codeaddr	2	50
JNZ	codeaddr	2	70
JZ	codeaddr	2	60
LCALL	codeaddr	3	12
LJMP	codeaddr	3	2
MOV	@R0, #data	2	76
MOV	@R0, A	1	F6
MOV	@R0, dataaddr	2	A6
MOV	@R1, #data	2	77
MOV	@R1, A	1	F7
MOV	@R1, dataaddr	2	A7
MOV	A, #data	2	74
MOV	A, @R0	1	E6
MOV	A, @R1	1	E7
MOV	A, dataaddr	2	E5
MOV	A, R0	1	E8
MOV	A, R1	1	E9
MOV	A, R2	1	EA
MOV	A, R3	1	EB
MOV	A, R4	1	EC
MOV	A, R5	1	ED
MOV	A, R6	1	EE
MOV	A, R7	1	EF
MOV	bitaddr, C	2	92
MOV	C, bitaddr	2	A2
MOV	dataaddr, #data	3	75
MOV	dataaddr, @R0	2	86
MOV	dataaddr, @R1	2	87
MOV	dataaddr, A	2	F5
MOV	dataaddr, dataaddr	3	85
MOV	dataaddr, R0	2	88
MOV	dataaddr, R1	2	89
MOV	dataaddr, R2	2	8A
MOV	dataaddr, R3	2	8B
MOV	dataaddr, R4	2	8C
MOV	dataaddr, R5	2	8D
MOV	dataaddr, R6	2	8E
MOV	dataaddr, R7	2	8F

(continued)

Mnemonic	Operand	Bytes	Hex Code
MOV	DPTR, #data	3	90
MOV	R0, #data	2	78
MOV	R0, A	1	F8
MOV	R0, dataaddr	2	A8
MOV	R1, #data	2	79
MOV	R1, A	1	F9
MOV	R1, dataaddr	2	A9
MOV	R2, #data	2	7A
MOV	R2, A	1	FA
MOV	R2, dataaddr	2	AA
MOV	R3, #data	2	7B
MOV	R3, A	1	FB
MOV	R3, dataaddr	2	AB
MOV	R4, #data	2	7C
MOV	R4, A	1	FC
MOV	R4, dataaddr	2	AC
MOV	R5, #data	2	7D
MOV	R5, A	1	FD
MOV	R5, dataaddr	2	AD
MOV	R6, #data	2	7E
MOV	R6, A	1	FE
MOV	R6, dataaddr	2	AE
MOV	R7, #data	2	7F
MOV	R7, A	1	FF
MOV	R7, dataaddr	2	AF
MOVC	A, @A+DPTR	1	93
MOVC	A, @A+PC	1	83
MOVX	@DPTR, A	1	F0
MOVX	@R0, A	1	F2
MOVX	@R1, A	1	F3
MOVX	A, @DPTR	1	E0
MOVX	A, @R0	1	E2
MOVX	A, @R1	1	E3
MUL	AB	1	A4
NOP		1	0
ORL	A, #data	2	44
ORL	A, @R0	1	46
ORL	A, @R1	1	47
ORL	A, dataaddr	2	15

*(continued)*

Mnemonic	Operand	Bytes	Hex Code
ORL	A, R0	1	48
ORL	A, R1	1	49
ORL	A, R2	1	4A
ORL	A, R3	1	4B
ORL	A, R4	1	AC
ORL	A, R6	1	4E
ORL	A, R7	1	4F
ORL	A, RS	1	4D
ORL	C, /bitaddr	2	A0
ORL	C, bitaddr	2	72
ORL	dataaddr, #data	3	43
ORL	dataaddr, A	2	42
POP	dataaddr	2	D0
PUSH	dataaddr	2	C0
RET		1	22
RET		1	32
RL	A	1	23
RLC	A	1	33
RR	A	1	3
RRC	A	1	13
SETB	bitaddr	2	D2
SETB	C	1	D3
SJMP	codeaddr	2	80
SUBB	A, #data	2	94
SUBB	A, @R0	1	96
SUBB	A, @R1	1	97
SUBB	A, dataaddr	2	95
SUBB	A, R0	1	98
SUBB	A, R1	1	99
SUBB	A, R2	1	9A
SUBB	A, R3	1	9B
SUBB	A, R4	1	9C
SUBB	A, R5	1	9D
SUBB	A, R6	1	9E
SUBB	A, R7	1	9F
SWAP	A	1	C4
XCH	A, @R0	1	C6
XCH	A, @R1	1	C7
XCH	A, dataaddr	2	C5

(continued)

Mnemonic	Operand	Bytes	Hex Code
XCH	A, R0	1	C8
XCH	A, R1	1	C9
XCH	A, R2	1	CA
XCH	A, R3	1	CB
XCH	A, R4	1	CC
XCH	A, R5	1	CD
XCH	A, R6	1	CE
XCH	A, R7	1	CF
XCHD	A, @R0	1	D6
XCHD	A, @R1	1	D7
XPL	A, R4	1	6C
XRL	A, #data	2	64
XRL	A, @R0	1	66
XRL	A, @R1	1	67
XRL	A, dataaddr	2	65
XRL	A, R0	1	68
XRL	A, R1	1	69
XRL	A, R2	1	6A
XRL	A, R3	1	6B
XRL	A, R5	1	6D
XRL	A, R6	1	6E
XRL	A, R7	1	6F
XRL	dataaddr, #data	3	63
XRL	dataaddr, A	2	62

# A2

## Development with DOS

---

While it is much easier to use an integrated environment such as  $\mu$ Vision, someone may want to develop software under DOS. Perhaps you do not have a PC that can run Windows®, or you feel that the  $\mu$ Vision approach insulates you too much from the development steps. The development environment invokes the individual program development tools for you, but the same software tools are running. Once you realize that, it seems to me, using DOS makes as much sense as hand assembling because it gives you a feel for how an assembler works! Nevertheless, I include the basic tools here for reference.

### **SOURCE CODE ENTRY**

Under DOS, you must come up with your own text editor. You can use most any word processor to edit source code, but some lack the special features related to computer program entry such as automatically beginning the next line at the same depth of indentation as the previous line. If you do use a word processor, be sure to save the file as a text-only file (with line feeds).

### **LOCATING CODE AND VARIABLES**

By default, the code will begin at location 0000 and the off-chip RAM variables will also begin at 0000. This is good for programming any system where the hardware is in the normal configuration.

If you are downloading your program with the use of a monitor program, your program goes into RAM during the development phases. Technically *code* should *never* go into RAM, but some logic ORing allows the *PSEN* and *RD* lines to address the same space and the monitor downloads by writing to the RAM as though it were data rather than code. Since the monitor uses some of the RAM for variables (as well as all the code space in its own EPROM), your code and variables must be located at different addresses from the monitor's and must not overlap each other. For the PU552, for example, downloaded code should start at  $2000_{16}$  and the off-chip (*XDATA* or *LARGE* model) variables should start after the code (usually at  $3000_{16}$  with the batch files). As is mentioned in the linking section, you need a space-reserving module to link in ahead of your program if your C program is to fall into the right places. In assembly, you can make your code an absolute module at the correct address.

When you are finished developing software by downloading, you may want to replace the monitor with your final code. Locate your code starting at  $0000_{16}$  (so power-up will start your program) and burn it in an EPROM which replaces the monitor EPROM. Because of the addressing on the PU552 board, however, the code addresses cannot overlap with the variables, so it is still necessary to force the *XDATA* to a higher address (usually  $2000_{16}$ ).

## COMPILING

The compiling actually involves a line like:<sup>1</sup>

---

```
C:>C51 [myfile.c51] [compiler directives]
```

---

In the absence of any directives, this will produce an object file, *myfile.obj* (used by the linker). It also produces a list file, *myfile.lst*, which adds line numbers, nesting depth, and any error or warning or error notes mixed in with the source text from the C file.

You can view files with a screen editor or word processor.<sup>2</sup> *.lst* files are the only way to identify errors (which you have to go back to the *source* to fix).

---

<sup>1</sup>Do not enter the square brackets [ ]. Instead, replace the whole construct with your specific information. For example, this line might be *C:>c51 step.c*.

<sup>2</sup>This is not necessary with  $\mu$ Vision where the errors lines are highlighted directly on the source file. To look at the results of your compiling or assembling, you can still *click File Open*, drop down the *List Files of Type:* menu and pick out *Listing Files (\*.LST; \*.M\*)*, and click on the file name of the source. Now my only use for list files is to examine the assembly code from the compiler when I need to know, in detail, what the compiler is doing. It is easier to see the same thing with the simulator (ds51) looking at the code in mixed mode.

If you wish, you can give the output files specific names, but usually you let them default to the same name as the source—*mypgm.LST* and *mypgm.OBJ*. The directives you might want to add include:

- *CODE* (to get the list file to include the resulting assembly code),
- *DEBUG* (to get the symbol names into the object file for later use by the simulator),
- *LARGE/COMPACT/SMALL* (the RAM model used by default),
- *ROM* (*large*, *small*, or *compact*, depending on the calls and jumps to be used for small programs or code for the 750/1), and
- *SYMBOLS* (to get a listing of all the symbols in the module).

There are a number of more directives to do with optimization, but these are the most commonly used ones.

## ASSEMBLING

In assembly you can produce relocatable code (with the *RSEG* directive) and handle it like the C code. It is more in beginning classes to simply begin the source with *ORG 2000H* (or whatever) to make absolute modules that do not need linking or locating. Then your entire processing would entail:

---

```
C:>ASM51 [myfile.a51]
C:>OHS51 [myfile.obj]
```

---

From the first line, the resulting file is *myfile.OBJ*.<sup>3</sup> You will get the *.lst* file, but the actual variable locations will be under your control when you do the source writing, and no *.m51* file will be produced. Remember that you will have to change the *ORG* and any memory pointers when you go from downloading with a monitor to burned code.<sup>4</sup>

---

<sup>3</sup>If you want to work out of another directory, you can use the *.set* command to have the operating system search other directories for the compiler. This is particularly useful if you are working off a floppy drive but using the tools on the hard drive.

<sup>4</sup>The assembler is invoked with the name of the assembler. I assume for the examples that follow that the tools are in C:\C51\BIN, where the Keil tools usually install themselves. Assuming your program is located in the same directory, you would type C:>*cd C:\c51\bin* to make the default path get to the assembler (and your source file). Then type C:>*asm51 myfile.A51* to assemble the file.



There are a number of directives you can add to the invocation line for the assembler. The only ones I think are important are the *DEBUG* (to get symbol information in the object file for use by the simulator) and *XREF* (to get the information in the list file about the various symbols):

---

```
C:>asm51 [myfile.A51] debug xref
```

---

If you have written relocatable assembly, you will have to follow the linking steps that follow, the same as you always do with compiled programs.

## LINKING

The linking now uses the “banked” linker, although you only need code banking for very large programs. Linking is necessary to combine multiple program modules of your own writing, but it is also necessary to include the C library modules if you use any C functions that aren’t handled directly by the compiler. Even if you do not need any other library functions, you can get the initialize module that sets all the variables to zero (as required by ANSI C). The only way to get a useful file without linking (and locating) is to write absolute assembly modules (using the *ORG* directive rather than relocatable segments). There are a lot of linker directives that can get quite complicated, but a typical linking line for a C program might look like:

---

```
C:>BL51 [myfile.obj],[libfiles.lib][drctvs]
```

---

This will produce a list file showing where all the symbols are located, *myfile.M51*, and an absolute, located file starting at zero, *myfile.myfile.m51*, shows specifically where all the variables and program pieces were actually located and holds any error messages from the linking process if external variables or functions could not be located or if code overlaps with absolute (usually assembly) modules. It is a good idea to check this file.

## HEX CONVERSION

The absolute file is in binary form, which is not useful for many of the PROM programmers and you cannot print it. The last step then is to convert the file to what is called Intel HEX format:

---

```
C:>OHS51 [myfile.obj]
```

---

This will produce *myfile.HEX*.

## BATCH FILES

There are several files developed by Dr. Richard Barnett for automating the compiling and linking process with Keil/Franklin C on the PU552 board. To avoid being dependent on "magic" you need to understand the process. If you understand the batch files, you will understand the changes needed when you get to the point of burning a program in EPROM. You can even develop your own set of commands to do what you want without so much typing.

Details of a few DOS batch files follow so you can see how they work. You would invoke the second one, for example, with *FCM552 MYFILE* to compile and link a C program you wrote named *MYFILE.C51* residing on the default path. The automatically included, space-reserving file you need, in that case, is *mon552.obj*.

---

### FA.BAT

```
echo off
c:\fc\bin\as51 %1.a51
```

---

---

### FCM552.BAT

```
echo off
c:\fc\bin\c51 %1.c51 SM debug NOIV pw(80)
c:\fc\bin\l51 c:\fc\inc\mon552.obj,%1.obj to %1
c:\fc\bin\ohs51 %1 HEX(%1.hex)
del %1
del %1.obj
```

---

---

### FCRAM2K.BAT

```
echo off
c:\fc\bin\c51 %1.c51 SM pw(80)c:\fc\bin\l51
c:\fc\inc\ram2k.obj,%1.obj to %1c:\fc\bin\ohs51 %1
HEX(%1.hex)
del %1
del %1.obj
```

---

---

**FCROM.BAT**

```

echo off
c:\fc\bin\c51 %1.c51 SM pw(80)
c:\fc\bin\l51 %1.obj
c:\fc\bin\ohs51 %1 HEX(%1.hex)
del %1
del %1.obj

```

---



---

**FCM51.BAT**

```

echo off
c:\fc\bin\c51 %1.c51 SM debug NOIV
pw(80)c:\fc\bin\l51 c:\fc\inc\mon.obj.%1.obj to %1
c:\fc\bin\ohs51 %1 HEX(%1.hex)
del %1
del %1.obj

```

---

**SPACE-RESERVING FILES**

If you are producing code for downloading to a board with a monitor, something additional must be done to force your code up at, for example, 2000<sub>16</sub>, and the off-chip data (xdata) at, for example, 3000<sub>16</sub>. This has been taken care of with a space-reserving file such as *mon552.obj*, but you can make your own using it as a model. The three listings below are the assembly for space-reserving files for the PU552 board and relatives.

---

**MON51.A51**

```

NAME ?C_STARTUP
PUBLIC ?C_STARTUP
PUBLIC GOMON
?STACK SEGMENT IDATA
RSEG ?STACK
    STACK: DS 1
XSEG AT 0
    EXTERN: DS 3000H
    EXTRN CODE(?C_START,EXT0,EXT1,CT0,CT1,SER)
CSEG AT 0
    ORG 3F03H
        LJMP EXT0 ;JUMP TO EXT INT 0
    ORG 3F0BH
        LJMP CT0 ;JUMP TO COUNT/TIMER 0 INT
    ORG 3F13H

```

```

    LJMP EXT1 ;JUMP TO EXT INT 1
    ORG 3F1BH
    LJMP CT1 ;JUMP TO COUNT/TIMER 1 INT
    ORG 3F23H
    LJMP SER ;JUMP TO SERIAL INT
    ORG 0000H
    DS 2000H ;SET CODE AT 2000H
    ?C_STARTUP: MOV SP,#LOW (STACK-1)
    LJMP ?C_START
    GOMON: MOV IF,#00
    LJMP 0030H
END

```

---

# **MON552.A51**

```

NAME ?C_STARTUP
PUBLIC ?C_STARTUP
PUBLIC GOMON
?STACK SEGMENT IDATAR
SEG ?STACK
    STACK: DS 1
XSEG AT 0
    EXTERN: DS 3000H
    EXTRN CODE (?C_START,X0,TM0,X1,TM1,S0,S1,
    CT0,CT1,CT2,CT3, ADC,CM0,CM1,CM2,TM2)
CSEG AT 0
    ORG 3F03H
    LJMP X0 ;JUMP TO EXT INT 0
    ORG 3F0BH
    LJMP TM0 ;JUMP TO TIMER 0 OVERFLOW
    ORG 3F13H
    LJMP X1 ;JUMP TO EXTERNAL INT 1
    ORG 3F1BH
    LJMP TM1 ;JUMP TO TIMER 1 OVERFLOW
    ORG 3F23H
    LJMP S0 ;JUMP TO S100(UART)
    ORG 3F2BH
    LJMP S1 ;JUMP TO S101(I^2C)
    ORG 3F33H
    LJMP CT0 ;JUMP TO T2 CAPTURE 0
    ORG 3F3BH
    LJMP CT1 ;JUMP TO T2 CAPTURE 1
    ORG 3F43H
    LJMP CT2 ;JUMP TO T2 CAPTURE 2

```

```

ORG 3F4BH
    LJMP CT3 ;JUMP TO T2 CAPTURE 3
ORG 3F53H
    LJMP ADC ;JUMP TO ADC COMPLETION
ORG 3F5BH
    LJMP CM0 ;JUMP TO T2 COMPARE 0
ORG 3F63H
    LJMP CM1 ;JUMP TO T2 COMPARE 1
ORG 3F6BH
    LJMP CM2 ;JUMP TO T2 COMPARE 2
ORG 3F73H
    LJMP TM2 ;JUMP TO T2 OVERFLOW
ORG 0000H    DS 2000H ;SET CODE AT 2000H
?C_STARTUP: MOV SP,#LOW (STACK-1)
    LJMP ?C_START
COMMON: MOV IE,#00
    LJMP 0080H
END

```

---

#### RAM2K.A51

```

NAME ?C_STARTUP
PUBLIC ?C_STARTUP
?STACK    SEGMENT    IDATA
RSEG ?STACK
    STACK: DS 1
XSEG AT 0
    EXTERN: DS 2000H
    EXTRN CODE (?C_START)
CSEG AT 0
    ?C_STARTUP: AJMP ?C_START
END

```

---

## LIBRARIES

You can *create* a new library, *add* or *replace* an object module in the library, or *delete* a module. You can also *list* the modules currently in the library. The librarian will refuse to add a new module if an existing module already has the same name or contains the same public symbol name. This avoids any confusion for the linker when it goes looking for unresolved external references. An example of use of the librarian is shown on the next page.

---

### Inserting into the Library

```
C:>C:\c51\bin\lib51
*CREATE A:myfns.lib
*ADD A:msecs.obj TO A:myfns.lib
*LIST A:myfns.lib
    MYFNS.LIB
    MSECS
*EXIT
```

---

The next step is to create a library to hold your module(s) (here it is named *myfns.lib*). In the first line, you enter the librarian (look for the \* prompt). Then you *CREATE* the library that will hold the module(s). Next, you *ADD* the module(s) to the library. To be sure the process has worked, an optional line is shown where you can *LIST* the current (new) contents of the library to be sure the new module is present. Finally, you leave the librarian with *EXIT* (not *quit*).

You can put public *variables* separately in the library if they are to be shared among modules—just make a module with variables and no functions.

The command line *list myfns.lib TO a:tmp.lst publics* shows any public function or symbol names and copies the result to a file—useful when the library gets too big to fit on one screen. Several functions *can* be added to a library under one module, but they will then *all* be brought into a calling program as a group if any one of them is called. That can eat up code space with unused functions. They will be treated as a single module in the library and listed as separate function names under the module name.

To use your new library, modify the linking line to include *myfns.lib* *after* your program as shown. Be sure that the link line is *all one line*. Insert a (*cr*) while editing the line if it is too long to show with the editor, but be sure to take it out before *using* the file.

---

```
BL51 sstart.obj,%1.obj,myfns.lib,c:\c51\lib\C51L.LIB
```

---

Suppose you make a mistake in the library module, and it does not work. The process of revision is straightforward. First, revise the C source file for the module and recompile it to get a new object module. Then put the new module in the library. First, however, *remove* the old module with the *DELETE* command. Several modules can be deleted at one time by listing the module names separated by commas. Use the *module* name as shown by the list command—*not* the name with the *.obj* extension and *not*

the name of the function inside the module unless it is the same as the module name. The function can have a different name than the module, as is shown.

---

#### Revieling a Library Module

```
c:>C:\c51\bin\lib51
*DELETE A:myfns.lib(msec)
*LIST A:myfns.lib
  MYFNS.LIB
*ADD A:msecs.obj TO A:myfns.lib
*LIST A:myfns.lib
  .MYFNS.LIB
    MSECS
*EXIT
```

---

# A3

## Language-switching Hints

---

### OTHER ASSEMBLY TO 8051 ASSEMBLY

Recognizing several fundamental differences should make the transition easier. First, there is no I/O space in the way that the 8085 defined it. There are the internal ports, defined as registers, but you address any added hardware as off-chip (*MOVX*) memory. The internal registers are *R0* through *R7* as well as *ACC* and *B*. Other than loading *DPTR*, there are no 2-byte loading instructions; it is much more an 8-bit machine than with the 8080/5 instruction set.

The @ symbol means indirect. You can only use *DPTR*, *R0*, and *R1* for such moves. Be careful to see that *MOV A,R0* is *not* the same as *MOV A,@R0*. The former brings the contents of *R0* into the accumulator whereas the latter brings the contents of the place *R0* points to into the accumulator. Another common error is the omitting of the # sign. *MOV A,#25* is quite different from *MOV A,25*; the former puts the binary equivalent of  $25_{10}$  into *ACC* whereas the latter puts the *contents* of address 25 ( $19_{16}$ ) into *ACC*. Notice that all these moves work only with internal RAM—there is only one way to get external RAM values—the *MOVX*.

With a good assembler, you can ignore the different kinds of calls and jumps. If you simply use *CALL* and *JMP*, the assembler will pick the smallest *SJMP* *AJMP* *LJMP* *ACALL* *LCALL* that will reach.

Notice that the zero flag reflects the current state of the accumulator at all times—you do not have to do a comparison instruction to get the flag set. However, there is only one comparison operation available. The *CJNE* is



quite powerful because it not only branches for inequality but it also sets the carry flag for the first byte being less. The *DJNZ* instruction makes looping quite efficient.

The only way to access code (EPROM) space is with the *MOVC* instruction, which is obviously set up for accessing lookup tables. It is impossible to write to code space unless a hardware OR of the *PSEN* and *RD* lines is included in the hardware.

The *MUL* and *DIV* instructions are new. They are good for only 8-byte operations and it is not clear if they do anything for multibyte math.

## PL/M TO C

This transition is not as difficult as it might seem. The structures are mostly the same and anything you can do in PL/M can also be done in C. Because the order of a few structures is reversed, you cannot do a totally automatic conversion by simple replacement. It is possible, however, to go *most* of the way with substitutions and then do some manual fix-up. Then you could condense the C a bit more with assignment operators and embedded assignments in *for* and *if* loops, but the result without the condensation is valid C and is probably more easily understood. It becomes a matter of style and choice although the condensed version may compile to slightly tighter code in some cases.

PL/M to C Substitutions	
DO:	{
END:	}
FOR:	for
IF	if
THEN	[nothing]
ELSE	else
DO WHILE	while
DO CASE	switch
=	= or ==
<>	!=
AND	& or &&
OR	or
NOT	~
XOR	^
MOD	%
BYTE	unsigned char
WORD	unsigned
MAIN	data (Keil/Franklin C)
AUXILIARY	xdata
CONSTANT	code
LITERALLY	#define

The table below lists substitutions and may help PL/M programmers change over. Remember that parentheses for arrays must go to square brackets and *if* and *for* blocks require parentheses. When you make multiple assignments to a common value (setting several variables to zero, for example) in PL/M you use commas while in C you use successive equals. The

comma in C has only a few uses such as in a *for* loop where you wish to do two things where you normally have only one. Where you declare several variables of the same type in PL/M you surround them with parentheses while in C you separate them with commas.

## STANDARD (ANSI) C TO C51

The changes involve *extensions* to C—mostly they specify the memory spaces you want used for variables. The memory model directives allow you to pick a default treatment.

You are probably used to having a console/printer and may miss the *printf()* and *scanf()* functions. For normal 8051 applications, there are no standard keyboard or printer connections and there may not be *any* device for such I/O! Most of the C51 compilers allow you to supply one or two primitive drivers to either go to an LCD module or a terminal (via the serial port) and come from either a keypad or a terminal. Nevertheless, for many embedded applications there is ultimately be no terminal involved, and it is foolish to try to make an 8051 into a personal computer!

The biggest change you face in switching from big-machine C to the 8051 is the change in *thinking*. It takes experience to avoid producing huge, inefficient programs due to requests for *float* and *trig* functions. Think in terms of simpler math or pre-computed lookup tables rather than coordinate transformations. The best C51 programmers move *up* from assembly where they have acquired a good knowledge of the hardware rather than *down* from standard C.

# A4

## Boards

---

Systems grow! I personally remember a project where I started out expecting that the code would fit in 4 K on-chip. At last report the code had gone off-chip and grown to use most of 64 K—the entire address space. I suspect the bulk of the code was not elaborate algorithms, but rather extensive user-prompt messages to send to the display. Nevertheless, the folks at Keil tell me that the majority of the applications they encounter are ones with code far beyond the 2 K that they offer in the compiler version with this book.<sup>1</sup>

Chapter 2 illustrates the addition of off-chip ROM, RAM, and I/O devices. When you hit the 64 K (16-bit addressing) limit for code you have two options. You can migrate to 16-bit processors such as Intel's x86 family, the 8096 family, or Motorola's 68000 family.<sup>2</sup> Alternatively, if you are reluctant to abandon your existing code and experience, you can go to *bank switching* where some port pins serve as additional address lines. I describe the hardware for that first.

---

<sup>1</sup>Obviously they hope that any commercial user will turn around and buy the version for larger code modules, although they assure me the included version is full-featured in every other respect.

<sup>2</sup>There are two families of code-compatible processor families now—the 251 family (Intel) and 8051EX family (Phillips).

Next I discuss two development boards—the MCB520 board from Keil with the Dallas 520 (available with an 8031 as well) and the PU552 board developed at Purdue using the ‘552.<sup>3</sup>

## BANK-SWITCHING CODE

What do you do if 64 K (sixteen address lines) is not enough? First you should look closely at code efficiency and your whole approach, but assuming that doesn't reveal any significant savings, you can go to bank switching. It involves having more than 64 K of total code with additional hardware to direct the code access to different blocks of code at different times. One to four port pins that are changed by software before a given bank is addressed supplement the sixteen address lines.<sup>4</sup> The process is best done while the program counter is in a space that does not change (called the common space—perhaps the lower 32 K). Then the various functions or code arrays that are switched in and out can be located in various 32 K upper banks.<sup>5</sup>

A schematic of a bank-switched design is shown next. If you look closely, this is a configuration with 512 K ( $64\text{ K} \times 8$ ) EPROMs so each one occupies the entire data space. The upper 3 bits of P1 control the bank selection.<sup>6</sup> The 74LS138 decodes this into the chip selects for the individual EPROMs. For power consumption reasons it is better to drive the *OE*/line from *PSEN* and let the decoder drive the *CS*/lines. That way the nonselected chips do not go through the internal fetching with the corresponding power drain.

---

<sup>3</sup>Dr. Richard Barnett, an associate of mine in the Electrical Engineering Technology Department (and author of *The 8051 Family of Microcontrollers* [1995 Prentice Hall, Englewood Cliffs, NJ (ISBN 0-02-306281-9)]) originally developed the 552 board. The board was revised by Rostek (John Rosheck) who is now handling the marketing and production (see Appendix A6 for addresses for these boards as well as a number of other suppliers).

<sup>4</sup>This is analogous to the four register banks in the *idata* space. When the software switches to a different bank it must first change 2 bits in the *PSW* register.

<sup>5</sup>It is possible to instead bank switch the entire 64 K space, given a duplicate image of the common code area at the same place in each bank. That may seem wasteful of EPROM, but with 27C512s being quite affordable, it can actually involve less hardware and cost as well as allowing for a smaller common space.

<sup>6</sup>Actually, with only four banks it could be done with 2 bits but I decided to show an expandable arrangement. This could hold eight banks—Keil/Franklin supports up to sixteen banks but somewhere you have to give up.



ture process requires no user-written code to do the switching. You only specify the number of banks possible, the port or xdata address of the bank control logic, and the first bit of the port controlling the bank selection. The linker manual describes the details of setting up the banks. For best efficiency you should apportion your functions so that frequently used ones are in the common area. The changing of banks takes about fifty processor cycles and eats up 2 bytes of stack, so you do not want to be switching unnecessarily. There are directives (`BANKn`, and `COMMON`) you can use within the modules or in invoking the linker/locator to determine where code functions or data will go.<sup>8</sup>

## PU552 MICROCONTROLLER BOARD

The PU552 board is a single-board computer intended for application in project development.<sup>9</sup> Features include:

1. The Phillips 83C552 microcontroller. The microcontroller features include:
  - 256 bytes of internal RAM
  - four 8-bit ports (+2 gone to off-chip expansion)
  - eight channels of 10-bit A/D
  - two pulse-width modulated outputs
  - A watch-dog timer
  - An extensive counter array.<sup>10</sup>
  - One serial port
  - multiprocessor serial communications (I<sup>2</sup>C bus)
2. The board's features additionally include:
  - An 8 K ROM (replaceable with a 32 K) in off-chip code space

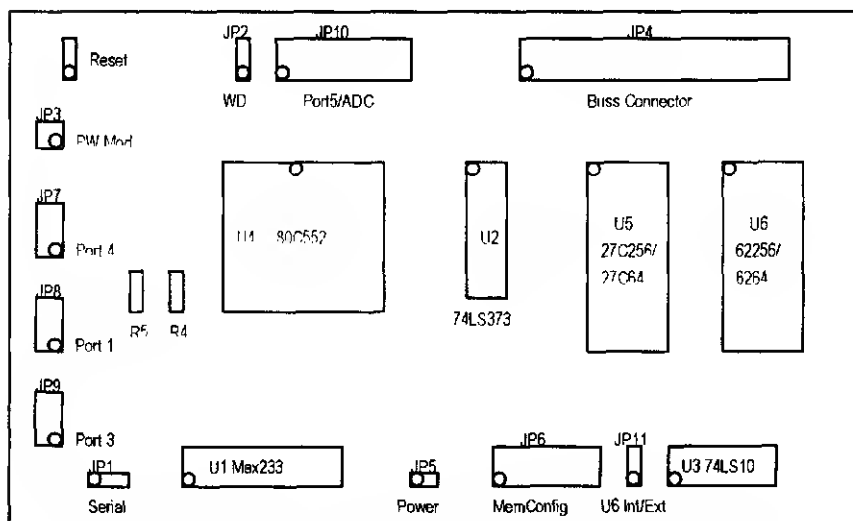
---

<sup>8</sup>Although the free Keil software includes a cobbled version of the banking linker, BL51, I think anyone doing bank switching will have exceeded the 2 K limit and should buy the complete version of the Keil/Franklin C compiler with the manuals—get more information there!

<sup>9</sup>This board's initial development was made possible by Robert and Helen Hoffer via the Hoffer Plastics Foundation who provided the seed money to produce the original set of boards. In addition, the Signetics Company initially donated the 552 microcontrollers and the INTEL Corporation donated the ROM memories. Dr. Richard H. Barnett, PE, designed the board. The monitor and test programs are likewise his work.

<sup>10</sup>For further details, refer to the 80C552 specification and application notes from Phillips/Signetics.

- An 8 K RAM (replaceable with a 32 K) which may either appear both in xdata *and* code space or xdata space only
  - One serial RS-232 port (converts the on-chip UART to RS-232)
  - Accessible *expansion bus connector* with all necessary address, data, and control lines
3. In order to use the board for software development you must have:
- A PC with an uncommitted serial port (COM2, usually)
  - A custom serial cable with a single-in-line connector on one end with transmit, receive, and ground signals<sup>11</sup>
  - A terminal emulator program or the freely available SEND31 program
  - Software development tools such as the C compiler and linker provided with this book



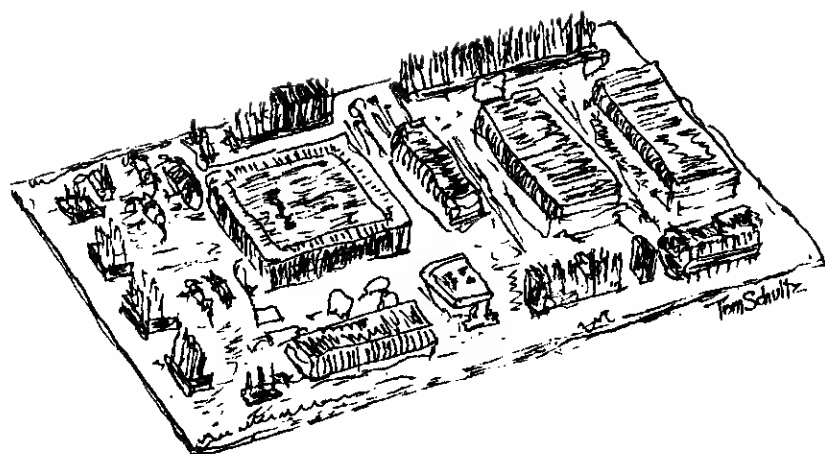
○ = Pin #1

**PU552-1 part layout (later versions have different layout)**

<sup>11</sup>You may prefer to make up a DB-9 or DB-25 socket from the PU552 board and use conventional serial cables.







**PU552 Sketch**

## HOOK-UP/CONNECTIONS

The PU552 requires regulated 5 VDC at 100 ma. Power is applied at JP5 as shown below:

**Power (JP5)**

Pin #	Function
1	Ground
2	Vcc

RS232 serial communications are accomplished via JP1. The baud rate is adjustable up to rates of 9600 baud. As configured "from the factory" (with the EET Monitor), the serial communication scheme is 9600 baud, 8 bits, no parity, and one stop bit. The pinout for JP1 is shown below:

**Serial Connection (JP1)**

Pin #	Function
1	Transmitted data
2	Ground
3	Received data

JP6 and JP11 configure the on-board memory. JP6 allows memory space to be configured for various size memory devices as well as to place the RAM device in external data space (only) or into both external code space and into external data space simultaneously (during program development).

JP11 allows the chip-select for U6 (off-chip RAM) to be controlled from off-board through JP4, the bus connector. *Because the current on-board memory-decoding scheme has large areas of fold-back, you must provide off-board decoding when adding additional devices.*

#### JP6 Jumpers (— indicates “factory” jumpers)

Pin #	Function
2-1	Development jumper. Places U6 in both code and data space
4-3	Jumper for 8 K RAM devices
5-6	Jumper for 32 K RAM devices
7-8	Jumper for constant U6 selection (leave open)
10-9	Jumper for 8 K ROM devices
11-12	Jumper for 32 K ROM devices
13-14	Jumper for 32 K RAM devices
15-16	Jumper for 8 K RAM devices

#### JP11 Jumper

Pin #	Function
2-1	Control U6 on-board

The available parallel ports of the 83C552, Port 1, Port 4, Port 5, and Port 3 are brought out on JP8, JP7, JP10, and JP9 respectively. Port 1 and Port 4 are usually used as 8-bit general-purpose I/O ports but they also have alternate functions relative to the counter arrays and to multiprocessor communications.<sup>12</sup> As with all 8051 family ports, no initialization is required. Port 3 is also a general purpose I/O port except that its bits have the same alternate functions as the 8051 (interrupts, serial communications, control lines, etc.). Port 5 is an input-only port which can provide either TTL level inputs or analog inputs to the A-D converter.<sup>13</sup> The port connections and functions are shown:

<sup>12</sup>The Port 1 and Port 4 alternate functions are described in the 80C552 specification and applications.

<sup>13</sup>The factory jumpering is shown. These jumpers are necessary to allow Port 5 to function as a TTL level input port or as a 0-5VDC analog input port. Other connections may be made to alter the A-D setup. See the 80C552 specification for details.

**Port 4 (JP7) and Port 1 (JP8)**

Pin #	Function
1	Bit 0 General purpose I/O
2	Bit 1 General purpose I/O
3	Bit 2 General purpose I/O
4	Bit 3 General purpose I/O
5	Bit 4 General purpose I/O
6	Bit 5 General purpose I/O
7	Bit 6 General purpose I/O
8	Bit 7 General purpose I/O

**Port 3 (JP 9)**

Pin #	Function
1	Bit 0 General purpose I/O or serial receive
2	Bit 1 General purpose I/O or serial transmit
3	Bit 2 General purpose I/O or interrupt 0
4	Bit 3 General purpose I/O or interrupt 1
5	Bit 4 General purpose I/O or timer 0 control
6	Bit 5 General purpose I/O or timer 1 control
7	Bit 6 General purpose I/O or external memory write
8	Bit 7 General purpose I/O or external memory read

**Port 5 bits and jumpering (JP10)**

Pin #	Function
1	Bit 0 General purpose Input or analog input ch 0
2	Bit 1 General purpose Input or analog input ch 1
3	Bit 2 General purpose Input or analog input ch 2
4	Bit 3 General purpose Input or analog input ch 3
5	Bit 4 General purpose Input or analog input ch 4
6	Bit 5 General purpose Input or analog input ch 5
7	Bit 6 General purpose Input or analog input ch 6
8	Bit 7 General purpose Input or analog input ch 7
10-9	Pin 9 is AVcc (analog Vcc) and is jumpered to pin 10 (Vcc)
12-11	Pin 11 is Vref+ and is jumpered to pin 12 (Vcc)
13-14	Pin 13 is AVss (analog ground) jumpered to pin 14 (GND)
15-16	Pin 15 is Vref- and is jumpered to pin 16 (GND)
17-18	Start ADC (STADC) is used to externally start a conversion

## OPERATION

As delivered, the PU552 will function with minimum setup. Connect the power pins to 5VDC and a serial cable to JPI. Connect your PC to the serial cable and start a communications program such as SEND31. Apply power to the PU552 and the monitor should sign on, ready for operation (in other words, PU> prompt should appear with sign-on message).<sup>14</sup>

You can test the PU552 by compiling and downloading the program shown here (you may need to modify the pathnames on the #include statements). This program will cause Port 4 to exactly duplicate the inputs on Port 5, and will blink alternate bits on Port 1 at a visible rate (it can be seen on LEDs—not provided). Likewise, the test program will echo any characters typed at the keyboard to the CRT and will exit to the monitor when a 0 (zero) is typed.

---

```

                /*Test Program for PU552 Board*/
#include <c:\c51\absacc.h>
#include <c:\c51\reg552.h>
void X0() interrupt 0 using 2 {}
void TM0() interrupt 1 using 2 {}
void X1() interrupt 2 using 2 {}
void TM1() interrupt 3 using 2 {}
void S0() interrupt 4 using 2 {}
void S1() interrupt 5 using 2 {}
void CT0() interrupt 6 using 2 {}
void CT1() interrupt 7 using 2 {}
void CT2() interrupt 8 using 2 {}
void CT3() interrupt 9 using 2 {}
void ADC() interrupt 10 using 2 {}
void CM0() interrupt 11 using 2 {}
void CM1() interrupt 12 using 2 {}
void CM2() interrupt 13 using 2 {}
void TM2() interrupt 14 using 2 {}
extern void gomon(); /*function to enter monitor*/

main(){
    unsigned int count;
    P1=0x55; /*set Port 1 starting value*/
    while(S0BUF != 0x30){/*loop while zero not typed*/
        P4=P5; /*echo Port 5 to Port 4*/
        count++; /*increment delay counter for LEDs*/
    }
}

```

---

<sup>14</sup>Some details of the EET monitor are covered in Chapter 9 beginning on page 249.

```

    if(count==10000){/*change LEDs if time elapsed*/
        count = 0; /*clear count for next cycle*/
        P1=~P1; /*complement Port 1*/
    }
    if(RI==1){/*check for serial character
        received*/
        RI=0; /*clear Receive flag*/
        SORBUF-SORBUF; /*echo character to screen*/
    }
}
gomon(); /*exit to monitor*/
}

```

---

## MCB520 EVALUATION BOARD

Keil Software, who provides the C compiler with this book, markets several boards including the one described here.

This is a considerably more elaborate board than the PU552. It can be run with a 87C51 or 87C52, but performs much faster with the Dallas 520 chip which runs instructions with fewer clock cycles and supports the second UART.

### 1. Features that distinguish this board from other 8051 boards:

- It includes 8 LFDs tied to port 1.
- It supports the 87C520's two serial ports with DB-9 cables. The MON51 program uses SIO1, so SIO0 is available for your programs.
- It runs directly off 5 V, but has an on-board voltage regulator and connector so it can run off, say, a 9 V wall transformer.
- Program development for most 8051-family target systems can use the same resources as the final version. The address decoding logic device relocates MON51 at 8000H on reset so downloaded programs can begin running at 0000H. The common serial port (0) is available during debugging because MON51 uses port 1 (in the 520 only).
- It has switch settings to support up to four code banks using a 256 Kx 8 EPROM or FLASH memory device. The code banking is directly supported by the compiler/linker which manages the banks by controlling address bits A16 and A17 from port pins (P1.6 and P1.7 respectively).
- It can support two 64 K byte banks of RAM (under direct user programming of port bit P1.5 to switch A16).

- It includes support for on-board programming of FLASH memory devices (*not* conventional EPROMS) including a 12 V DC-DC converter.
2. Equipment supplied with the MCB520:
    - The board with an installed MON51 ROM (MON51.HEX).
    - A 9 V wall transformer power supply (or other source of 9 V @ 500mA)
    - A disc with Keil MON51 terminal program (MON51.EXE)
    - A User's Guide with far more information than is provided here
  3. Equipment you must supply:
    - A PC host with Windows 3.1, 3.11, or Win 95
    - A serial cable—straight DB-9 male to DB-9 female
    - An EPROM programmer if you intend to burn finished programs into EPROM (rather than FLASH) memory

### Switch Settings for the MCB520 Board

#### *JPI*

1-2 (default) 28-pin (up to 512K) ROM device

2-3 32-pin (1Meg) ROM installed using address bit A17

**SWI** (note: numbers are silk-screening numbers—backwards from pin order on schematic drawing) *default is in bold*

Name	Number	Off (open)				On (closed)	
E17	SW1-1	P1.7 is available				P1.7 can drive ROMA17 (see JPI)	
E16	SW1-2	P1.6 is available and ROMA16 is grounded				P1.6 drives ROMA16	
R16	SW1-3	RAMA16 is grounded				P1.5 drives RAMA16	
FOK	SW1-4	P3.3 is available					
FEN	SW1-5	P1.4 is available				P1.4 (low) enables flash programming voltage	
EA	SW1-6	starts up with code in CPU				starts up with code in EPROM/FLASH	
HRM	SW1-7	off	code and xdata separate	off	EPROM 0-7FFF maps to code RAM 0-7FFF maps to xdata	on	EPROM 0-7FFF maps to code at 8000 to Ffff
VNM	SW1-8	off		on	RAM 8000 to FFFF maps to code and xdata	on	RAM 0-7FFF maps to code and xdata

**SW2** (note: numbers are silk-screening numbers—backwards from pin order on schematic drawing) *default is in bold*

Name	number	off (open)	on (closed)
LD	SW2-1	LEDs off port 1 do not light	LEDs off port 1 light
PR	SW2-2	header pin 40 floats	header pin 40 tied to Vcc
3.2	SW2-7	INT push button not connected to P3.2	INT push button connected to P3.2
3.4	SW2-8	INT push button not connected to P3.4	INT push button connected to P3.4
T1	SW2-3	no connect to P1.3	P1.3 drives MAX232
R1	SW2-4	no connect to P1.2	MAX232 drives P1.2
T0	SW2-5	no connect to P3.1	P3.1 drives MAX232
R0	SW2-6	no connect to P3.0	MAX232 drives P3.0

## Writing Programs for the MCB520 Board

The procedure is similar to that of the PU552 with a few exceptions:

- *Do not write to P1.2 or P1.3*—if you do you will trash the communication between the board and the host PC (because it uses serial port 1—only available on the Dallas chips—which ties up those pins).
- With the default settings, *MON51* code is relocated to 8000H so you can (and should) *start your target program at 0000H*—you do not need to relocate your program during development.
- If you have the board with the 520 chip (some versions come with other processors), the accompanying monitor is configured for the 33 Mhz clock. If you change it, the serial communication will not work from *MON51*.<sup>15</sup>
- You can configure dScope to enable serial breaks. In this case *MON51* overwrites the serial port 1 communications interrupt (INT7) to point to the *MON51*. Since your target program could occupy that space, either disable serial breaks or *add a dummy interrupt* to reserve the space:

---

```
static void dummy_isr interrupt 7{}
```

---

The 87C520 CPU as provided has 16 K of OTP (one-time-programmable) memory that holds *MON51*.

---

<sup>15</sup>The provided monitor, *MON51*, is briefly described in Chapter 9 beginning on page 256. Much more information is supplied when you purchase the board.

### Internal Port Availability

As you may realize, when the 8051-family is expanded, the port availability decreases. Adding off-chip ports was discussed earlier in the pages on expanding, but the table below summarizes the port situation, which is highly dependent on the option switch settings.

Port pin	Usage	Description
P0.0-P0.7	AD0-AD7	Lower address multiplexed with data to off-chip memory
P1.0	free	
P1.1	free	
P1.2	RXD1	When R1 is on, connects to the MAX232 to SIO1
P1.3	TXD1	When T1 is on, connects to the MAX232 to SIO1
P1.4	FEN	When FEN switch is on a low on this pin enables the 12v flash programming supply
P1.5	RAMA16	When R16 is on, drives A16 on the RAM socket
P1.6	ROMA16	When E16 is on, drives A16 on the ROM socket
P1.7	ROMA17	When E17 is on, drives A17 on the ROM socket
P2.0-P2.7	A8-A15	Upper address bits to off-chip memory
P3.0	RXD0	When R0 is on, connects to the MAX232 to SIO0
P3.1	TXD0	When T0 is on, connects to the MAX232 to SIO0
P3.2	INT BUTTON	When 3.2 switch is on this pin (INT0—external interrupt input) driven from INT button
P3.3	FOK	When 1 OK switch is on, receives signal indicating flash programming voltage
P3.4	INT BUTTON	When 3.4 switch is on this pin (T0—the counter input) is driven from the INT button
P3.5	free	
P3.6	WR/	Control signal for off-chip RAM
P3.7	RD/	Control signal for off-chip RAM



The MCB520 Evaluation Board Schematic is below.



## Other Commercial Boards

Although it is quite possible to wire-wrap an 8051 prototype system, it is probably more economical for commercial project development to make use of pre-built boards on the market. Most of them come with a monitor program to download code from a host (PC) to RAM by way of a serial port. That's the way I tested most of the examples in this book.

One drawback of developing with a commercial board and a monitor program is the fact that the monitor takes up the resources to communicate to the host development PC as well as the code and RAM space. They are unavailable while developing your application program. The economics of a monitor are usually better than using an in-circuit-emulator,<sup>16</sup> but you have to decide what the cost goals for the final product are as well as the funds available for development tools. The commercial boards have the advantage of arriving already tested with some degree of documentation. There are apparently products with directly connecting add-on boards for A-D, LCD/keyboard interface, etc. Such hardware could save significant design and debugging time!

The list of board suppliers in Appendix A6 will never be final as companies and designs come and go, but it provides some idea of the sorts of products available. Because the 8051 design is mature, the only expected changes are designs that incorporate the newer members of the 8051 family.

---

<sup>16</sup>See Chapter 9 for a little on in-circuit emulators.

# A5

## The 8051 Family

---

The family of 8051 relatives continues to grow, to the point where the core is even available for ASIC design libraries. IC designers can put the processor on-chip with other circuitry to create a truly custom device. Most applications probably do not have the volume to justify such cost yet, but automated design tools may be changing that.

### SPECIAL FEATURES

A few features of the extended family are different enough to merit brief descriptions.

**Pulse width modulated (PWM) outputs** are quite useful in driving DC motors at variable speed. You run the motor with a series of constant-frequency pulses; where, the *wider* the pulses, the faster the motor turns.

By adding an integrating circuit,<sup>1</sup> PWM outputs can provide analog outputs. With the 80C552, for example, three registers control the two PWM outputs. One determines the repetition rate for both outputs and the other two registers determine the width of the respective outputs. The width is broken up into a resolution of 255 and the repetition rate can be from 92 Hz to about 23 kHz.

Use the **watchdog timer** to reset the microcontroller if it enters an error state (perhaps through noise or programming errors). Your program must get back to restart the time-out before the time expires or the timer will cause a reset of the processor. It requires that you determine how long you can allow the processor to be "lost" and how often you can get around to restarting the timer. There is a complicated enough reloading process to make it unlikely that random code would restart the timer. It can be enabled by tying a pin low

---

<sup>1</sup>This can be as simple as a resistor and a capacitor.

and cannot be disabled in software. Particularly if your system has EEPROM, it may be desirable to recognize the difference between a “cold” and “warm” start to enable a smoother restart from where things left off.

You need *high-speed capture and compare* logic for precise timing beyond that possible with program flow. An external signal can latch the count. You can then measure short time intervals more accurately. The compare logic can initiate timed interrupts and also provide precisely timed outputs.

Symbol	Description	Direct address
SADDR	Slave address	A9 <sub>16</sub>
SADEN	Slave address mask	B9 <sub>16</sub>
RACAP2L	Timer 2 capture low	CA <sub>16</sub>
RACAP2H	Timer 2 Capture High	CB <sub>16</sub>
T2CON	Timer 2 Control	CS <sub>16</sub>
T2MOD	Timer 2 Mode Control	C9 <sub>16</sub>
TL2	Timer Low 2	CD <sub>16</sub>
TH2	Timer High 2	CC <sub>16</sub>
CCON	PCA Counter Control	D8 <sub>16</sub>
CMOD	PCA Counter Mode	D9 <sub>16</sub>
CCAPM0	Module 0 Mode	DA <sub>16</sub>
CCAPM1	Module 1 Mode	DB <sub>16</sub>
CCAPM2	Module 2 Mode	DC <sub>16</sub>
CCAPM3	Module 3 Mode	DD <sub>16</sub>
CCAPM4	Module 4 Mode	DE <sub>16</sub>
CL	PCA Counter Low	E9 <sub>16</sub>
CCAP0L	Module 0 Capture Low	EA <sub>16</sub>
CCAP1L	Module 1 Capture Low	EB <sub>16</sub>
CCAP2L	Module 2 Capture Low	EC <sub>16</sub>
CCAP3L	Module 3 Capture Low	ED <sub>16</sub>
CCAP4L	Module 4 Capture Low	EE <sub>16</sub>
CH	PCA Counter High	F9 <sub>16</sub>
CCAP0H	Module 0 Capture High	FA <sub>16</sub>
CCAP1H	Module 1 Capture High	FB <sub>16</sub>
CCAP2H	Module 2 Capture High	FC <sub>16</sub>
CCAP3H	Module 3 Capture High	FD <sub>16</sub>
CCAP4H	Module 4 Capture High	FE <sub>16</sub>

**Additional specisi function registers for 80C51FA and FB<sup>1</sup>**

Symbol	Description	Direct address
<i>P0</i>	<i>Port 0</i>	<i>80<sub>16</sub></i>
SP	Stack pointer	81 <sub>16</sub>
DPL	Data pointer low	82 <sub>16</sub>
DPH	Data pointer high	83 <sub>16</sub>
PCON	Power control	87 <sub>16</sub>
<i>TCN</i>	<i>Timer control</i>	<i>88<sub>16</sub></i>
TMOD	Timer mode	89 <sub>16</sub>
TL0	Timer low 0	8A <sub>16</sub>
TL1	Timer low 1	8B <sub>16</sub>
TH0	Timer high 0	8C <sub>16</sub>
TH1	Timer high 1	8D <sub>16</sub>
<i>P1</i>	<i>Port 1</i>	<i>90<sub>16</sub></i>
SOCON	Serial 0 control	98 <sub>16</sub>
SOBUF	Serial 0 data buffer	99 <sub>16</sub>
<i>P2</i>	<i>Port 2</i>	<i>A0<sub>16</sub></i>
<i>IEN0</i>	<i>Interrupt enable 0</i>	<i>A8<sub>16</sub></i>
CML0	Compare low 0	A9 <sub>16</sub>
CML1	Compare low 1	AA <sub>16</sub>
CML2	Compare low 2	AB <sub>16</sub>
CTL0	Capture low 0	AC <sub>16</sub>
CTL1	Capture low 1	AD <sub>16</sub>
CTL2	Capture low 2	AE <sub>16</sub>
CTL3	Capture low 3	AF <sub>16</sub>
<i>P3</i>	<i>Port 3</i>	<i>B0<sub>16</sub></i>
<i>IP0</i>	<i>Interrupt priority 0</i>	<i>B8<sub>16</sub></i>
P4	Port 4	C0 <sub>16</sub>
P5	Port 5	C4 <sub>16</sub>
ADCON	ADC control	C5 <sub>16</sub>
ADCH	AD converter high	C6 <sub>16</sub>
<i>TM2IR</i>	<i>Timer 2 int flag reg</i>	<i>C8<sub>16</sub></i>
CMH0	Compare high 0	C9 <sub>16</sub>
CMH1	Compare high 1	CA <sub>16</sub>
CMH2	Compare high 2	CB <sub>16</sub>
CTH0	Capture high 0	CC <sub>16</sub>

<sup>1</sup>In addition to the standard 8051 SFRs found in Chapter 2, page 21. Italics SFRs are also bit-addressable.

**Complete 8XC552 special function registers<sup>†</sup>**

CTH1	Capture high 1	CD <sub>16</sub>
CTH2	Capture high 2	CE <sub>16</sub>
CTH3	Capture high 3	CF <sub>16</sub>
<i>PSW</i>	<i>Program status word</i>	<i>D0<sub>16</sub></i>
<i>SICON</i>	<i>Serial 1 control</i>	<i>D8<sub>16</sub></i>
S1STA	Serial 1 status	D9 <sub>16</sub>
SIDAT	Serial 1 data	DA <sub>16</sub>
SIADR	Serial 1 address	DB <sub>16</sub>
<i>ACC</i>	<i>Accumulator</i>	<i>E0<sub>16</sub></i>
<i>IEN1</i>	<i>Interrupt enable 1</i>	<i>E8<sub>16</sub></i>
TM2CON	Timer 2 control	EA <sub>16</sub>
CTCON	Capture control	EB <sub>16</sub>
TML2	Timer low 2	EC <sub>16</sub>
TMH2	Timer high 2	ED <sub>16</sub>
STE	Set enable	EE <sub>16</sub>
RTE	Reset/toggle enable	EF <sub>16</sub>
<i>B</i>	<i>B register</i>	<i>F0<sub>16</sub></i>
IP1	Interrupt priority 1	F8 <sub>16</sub>
PWM0	PWM register 0	FC <sub>16</sub>
PWM1	PWM register 1	FD <sub>16</sub>
PWMP	PWM prescaler	FE <sub>16</sub>
T3	Timer 3	FF <sub>16</sub>

<sup>†</sup>This is the chip used on the PU552 board. The SFRs in italics are also bit-addressable

## SPECIAL SERIAL COMMUNICATION FEATURES

In addition to the shift-register and ninth-bit modes of the standard 8051, there is another serial option that deserves attention, the I<sup>2</sup>C bus of the '552, '751, and '752.<sup>2</sup> The I<sup>2</sup>C bus was developed by Signetics/Philips and is an open-collector, wired-OR bus, which allows multiple masters with a resolution of bus contention. It is quite slow by serial bus standards (only up to about 100k bits per second), but it is amenable to much slower devices. It involves two lines, *SCL* and *SDA*—the clock and data respectively. Without getting into details here, data transfer occurs when the master of the moment drives the clock line. If the master is sending, then it also drives the data line. Otherwise, the replying slave drives the data line in synchronism with the clock driven by the master. If the master is clocking too fast, the slave can hold the clock low to make it wait. That is where the wired-OR feature comes in handy. There are details relating to start and stop conditions as well as arbitration when several devices attempt to become the master at the same time.

The I<sup>2</sup>C protocol can be generated with normal port lines on processors that do not have direct support. The data rate can be slow enough to allow it to be easily tracked in software. Aside from communication between processors, the protocol is attractive because there are additional I<sup>2</sup>C peripheral devices such as small RAM and EEPROMs, D-A converters, port expanders, LCD drivers, voice and frequency synthesizers, and several radio and telecommunication devices.

The table that follows lists, in a numeric sequence, *all* known 8051 relatives. Most members have versions without on-chip factory-masked ROM (usually with a 3 in the number as 8031) and a few have EPROM on chip

---

<sup>2</sup>I should also mention the SDLC controller in the 8044 family, which may not be in production much longer. It seems odd to describe the '44 as a *special* member of the family because it dates back almost to the start of the 8051. (You may have heard of the 8041 and 8042, which were the parallel-port relatives not based on the 8051 core.) But the '44 was the heart of the BITBUS protocol, mentioned in Chapter 13. The BITBUS/DCX system has been de-emphasized in this edition because it has dropped out of Intel's interest. Basically the 8044 family was an 8051 core with the UART replaced with a SDLC (IBM's Synchronous Data Link Control protocol) controller. The serial interface unit handled all the details of serial communication: zero bit insertion/deletion, address recognition, CRC (Cyclic Redundancy Check), and frame sequence check were all done automatically. By handling all the communication, the serial interface unit freed the CPU to concentrate on other tasks. When a message arrived or was sent, the transfer occurred in a shared memory area above address 128. With BITBUS this communication was the heart of *distributed* control. The protocol allowed sending of lengthy blocks of data with addresses, length indication, and an error-checking code (somewhat like a parity bit on a byte). The protocol was quite fast (up to 2.4 MHz) and had a good degree of error checking.

(usually with a 7 in the number as 8751). The blank spaces in the table only indicate that specific data was not located. This table is only hints at the capabilities available. The data sheets from the manufacturers have more details. For example, the FA...FC versions have a host of modes and up to five outputs from the fourth timer, which make them quite flexible, but which are too complex to describe in a table like this.



DEVICE	On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)	On-chip RAM	I/O Port Pcs	Timer/counters	ext interrupts	# Analog in /bits	Mfg. (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Matra)	# pins	Max. Speed Avail. (MHz)	Key features: many have memory lock, power save modes. Unless noted, all are CHMOS process
80C31	0	128	32	2	2		M,P	40, 44	33	Industry Standard
8031AH	0	128	32	2	2		I	40, 44	12	HMOS
80C31BH	0	128	32	2	2		I	40, 44	16	Power Save Modes
80C31	0	128	32	2	2		P	40, 44	16	Low Voltage/Power (1.8V to 6V)
080C31-L	0	128	32	2	2		M	40, 44	6	2.7/10 6V, Low Power
80C31u-L	0	128	32	2	2		M	40, 44	6	2.7/10 6V, Low Power
80C31u	0	128	32	2	2		M	40, 44	42	
80C32	0	256	32	3	2		I,M,P,S	40, 44	40	Industry Standard
8032AH	0	256	32	3	2		I	40, 44	12	HMOS
80C32-L	0	256	32	3	2		M	40, 44	6	2.7/10 6V, Low Power
80C32u	0	256	32	3	2		M	40, 44	42	
80C32u-L	0	256	32	3	2		M	40, 44	16	2.7/10 6V, Low Power
8XC51	4K	128	32	2	2		I,M,P	40, 44	33	Industry Standard
8XC51AH	4K	128	32	2	2		I	40, 44	12	HMOS
80C51BH	4K	128	32	2	2		I	40, 44	16	Power Save Modes
8751BH	4K	128	32	2	2		I	44	12	HMOS
80C51-L	4K	128	32	2	2		M	40, 44	6	2.7/10 6V, Low Power

<i>Key features: many have memory lack, power save modes. Unless noted, all are CHMOS process</i>	<i>Max. Speed Avail. (MHz)</i>	<i># pins</i>	<i>Mfg. (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Mntrn)</i>	<i># Analog in/bits</i>	<i>ext interrupts</i>	<i>Timer/counters</i>	<i>I/O Port Pins</i>	<i>On-chip RAM</i>	<i>On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)</i>	<i>DEVICE</i>
	42	40, 44	M		2	2	32	126	4K	80C51u
2.7/10 6V, Low Power	16	40, 44	M		2	2	32	128	4K	80C51u-L
*flash EPROM	24	40, 44	A		2	2	32	128	4K*	89C51
2.7-6V, *flash EPROM	12	40, 44	A		2	2	32	128	4K*	89LV51
Prog. Counter Array (PCA)	24	40, 44	LP		2	3	32	256	8K	87C51FA
Prog. Counter Array (PCA) 2.7-3.6V	20	44	LP		2	3	32	256	8K	8XL51FA
Prog. Counter Array (PCA)	24	40, 44	LP		2	3	32	256	16K	8XC51FB
Prog. Counter Array (PCA) 2.7-3.6V	20	44	LP		2	3	32	256	16K	8XL51FB
Prog. Counter Array (PCA)	24	40, 44	LP		2	3	32	286	32K	8XC51FC
Prog. Counter Array (PCA) 2.7-3.6V	20	44	I		2	3	32	256	32K	8XL51FC
4 Ch. 8-bit A/D, 2 PCA, 6 I/O ports	16	44, 68	I	8	2	3	48	256	8K	8XC51GB
Customized Keyboard Ctrl., Power Save Modes	16	100	I	8	2	2	87	256	8K	8XC51SL-BG
4 Ch. 8-bit A/D, Power Save Modes	16	100	I	8	2	2	87	256	16K	8XC51SL-AH
Low voltage, Power Save	16	100	I	8	2	2	87	256	16K	8XC51SL-AL
	40	40, 44	L.M.P.S		2	3	32	256	8K	8XC52
HMOS	12	40, 44	I		2	3	32	256	8K	8752BH

Key features may have memory lack, power save modes. Unless noted, all are CMOS process	Max. Speed Avail. (MHz)	# pins	Mfg., (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Matra)	# Analog in/bits	ext interrupts	Timer/counters	I/O Port Pins	On-chip RAM	On-chip ROM (X: 0 ROM, 3 ROMLESS, 7 EPROM)	DEVICE
	42	40, 44	M		2	3	32	256	8K	80C52u
2.7 to 6V, Low Power	6	40, 44	M		2	3	32	256	8K	80C52u-L
2.7 to 6V, Low Power	16	40, 44	M		2	3	32	256	8K	80C52u-L
flash EPROM	24	40, 44	A		2	3	32	256	8K*	89C52
2.7-6V, *flash EPROM	12	40, 44	A		2	3	32	256	8K*	89L152
Standard 8XC51 Compatible	16	40, 44	I.P.		2	3	32	256	16K	8XC54
2.7-3.6V, *OTP	20	44	I		2	3	32	256	16K*	8XL54
Reduced EMI, OSD, 9 PWM Outputs	20	42	P		2	2	28	256	16K	8XC055
Standard 8XC51 Compatible	24	40, 44	I.P.		2	3	32	256	32K	8XC58
2.7-3.6V, *OTP	20	44	I		2	3	32	256	32K*	8XL58
Multi-protocol Serial comm., Power save	16.5	44, 48	I		2	2	40	256	8K	8XC152JA
Multi-protocol Serial comm., Power save	16.5	44	I		2	2	56	256	8K	80C152JB
WD	16	40, 44	M		2	3	32	256	16K	8XC154
2.7 to 6V, Low Power, WD	6	40, 44	M		2	3	32	256	16K	8XC154-L
WD	42	40, 44	M		2	3	32	256	16K	8XC154u
2.7 to 6V, Low Power, WD	16	40, 44	M		2	3	32	256	16K	8XC154u-L

<i>Key features many have memory lock, power save modes. Unless noted, all are CHMOS process</i>	<i>Max. Speed Avail. (MHz)</i>	<i># pins</i>	<i>Mfg., (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Matra)</i>	<i># Analog in/bits</i>	<i>ext interrupts</i>	<i>Timer/counters</i>	<i>I/O Port Ppins</i>	<i>On-chip RAM</i>	<i>On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)</i>	<i>DEVICE</i>
WD	16	40, 44	M		2	3	32	256	32K	83C154D
2.7 to 6V, Low Power, WD	6	40, 44	M		2	3	32	256	32K	83C154D-L
WD	42	40, 44	M		2	3	32	256	32K	83C154Du
2.7 to 6V, Low Power, WD	16	40, 44	M		2	3	32	256	32K	83C154Du-L
OSD, 9 PWM Outputs, 8 LED Drivers I2C (2)	8	64	P	4	13	3	47	256	16K	83CL167
OSD, 9 PWM Outputs, 8 LED Drivers I2C (2)	8	64	P	4	13	3	47	256	16K	83CL168
OSD, 9 PWM Outputs, 8 LED Drivers I2C (2)	8	64	P	4	13	3	47	256	12K	83CL267
OSD, 8 PWM Outputs, 3 wire Serial I/O I2C(2)	8	64	P	4	13	3	47	256	12K	83CL268
2 DPTRs	-75	40, 44	D		6	3	32	256	0	80C310
WD, 2 DPTRs, pwr fail detect	-75	40, 44	D		6	3	32	256	0	80C320
WD, 2.7-5.5V, 2 DPTRs, pwr fail detect	-75	40, 44	D		6	3	32	256	0	80C323
Low Voltage/Power(1.8V-6V) I2C Bus	12	40	P		10	2	32	128	4K	8XC1410
Processor Bus Interface	16	68	P		2	2	56	128	4K	8XC451

<i>Key features many have memory lock, power save modes. Unless noted, all are CHMOS process</i>	<i>Max. Speed Avail. (MHz)</i>	<i># pins</i>	<i>Mfg., (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Matra)</i>	<i># Analog in/bits</i>	<i>ext interrupts</i>	<i>Timer/counters</i>	<i>I/O Port PPIs</i>	<i>On-chip RAM</i>	<i>On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)</i>	<i>DEVICE</i>
	40	40, 44	S		2	3	32	256	8K	80C501
WD (2)	18	40, 44	S		2	3	32	512	16K	80C502
WD (2)	18	40, 44	S	10	2	3	32	256	8K	80C503
8XC54 with Hardware Divide	20	40, 44	P		2	2	32	256	16K	8XC504
NMOS, WD	12	68	S	8/10	12	3	48	256	8K	80515
WD	16	68, 80	S	8/10	12	3	56	256	8K	8XC515
WD	18	68, 80	S	10	12	3	56	1.2K	32K	8XC515A
WD (2)	16	84, 100	S	8	14	4	68	256	8K	80C517
WD (2)	18	84, 100	S	10	17	4	68	2.2K	32K	8XC517A
WD	12	68	S	8	12	3	48	256	0	80535
WD (2)	16	84, 100	S	8	14	4	68	256	0	80C537
WD, 2 DPTRs, pwr fail detect	-75	40, 44	D		6	3	32	1.2K	16K	83C520
WD, 2 DPTRs, pwr fail detect	-75	40, 44	D		6	3	32	1.2K	16K	87C520
WD I2C Serial Bus	20	40, 44	P		2	3	32	512	16K	8XC524
Large memory, WD I2C Bus	20	40, 44	P		2	3	32	512	32K	8XC528
Reduced EMI version of 8XC528	16	44	P		2	3	32	512	32K	8XCE528
WD, 2 DPTRs, pwr fail detect	-75	52	D		6	3	32	1.2K	16K	87C530

<i>Key features many have memory lock, power save modes. Unless noted, all are CHMOS process</i>	<i>Max. Speed Avail. (MHz)</i>	<i># pins</i>	<i>Mfg., (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Matra)</i>	<i># Analog in/bits</i>	<i>ext interrupts</i>	<i>Timer/counters</i>	<i>I/O Port Pins</i>	<i>On-chip RAM</i>	<i>On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)</i>	<i>DEVICE</i>
8042 Compatible, ACCESS bus, I <sup>2</sup> C Bus  WD  PWM, WD, T2 I <sup>2</sup> C Serial Bus  32K FLASH, WD, PWM, Low EMI, I <sup>2</sup> C  PWM, WD, T2  LV detect, Osc. detect, low EMI, 4 comp  LV detect, Osc. detect, low EMI, 4 comp, UPI  WD, PWM, Low Voltage (2.5V to 6V) I <sup>2</sup> C  CAN Bus, WD, T2  CAN bus, WD, T2, PWM, reduced EMI  I <sup>2</sup> C Serial Bus  16K ROM/EPROM, I <sup>2</sup> C Serial Bus  8XC654 with Reduced EMI  PWM  24-Pin Skinny DIP	16	44	P		2	2	8	256	4K	8XC542
	16	40, 44	P	8	2	2	32	128	4K	8XC550
	30	68, 80	P	10	2	3	48	256	8K	8XC552
	16	80	P	10	6	3	48	1024	32K	8XC558
	16	68, 80	P	8	2	3	48	256	8K	8XC562
	16	40, 44	P		2	3	32	256	8K	8XC575
	16	40, 44	P	10	2	3	32	256	8K	8XC576
	12	56, 64	P	8	10	3	4	256	6K	8XCL580
	16	68	P	1	6	3	48	512	16K	8XC592
	16	80	P	10	6	3	48	512	32K	8XC598
	24	40, 44	P		2	2	32	256	8K	8XC632
	24	40, 44	P		2	2	32	256	16K	8XC654
	16	44	P		2	2	32	256	16K	8XC654
	16	24, 28	P		2	2	19	64	8K	8XC748
	16	28	P	8	2	2	21	64	8K	8XC749
	40	24, 28	P		2	1	19	64	8K	8XC750

DEVICE	8XC751	8XC752	83CL781	83CL782	8XC851	83C852	89C1051	89C2051	89S8252	DS5000	DS5001	DS5002
On-chip ROM (X: 0 ROM 3 ROMLESS 7 EPROM)	2K	2K	16K	16K	4K	6K	1K*	2K*	8K*	8K-64K	32K-218K	32K-128K
On-chip RAM	64	64	256	256	128	256	64	128	256	**	**	**
I/O Port Pciens	19	21	32	32	32	2	15	15	32	32	32	32
Timer/counters	1	1	3	3	2	2	1	2	3	2	2	2
ext interrupts	2	2	10	10	2	1	2	2	2	2	2	2
# Analog in/bits		8										
Mfg., (A=Atmel, D=Dallas, I=Intel, P=Phillips, S=Siemens, M=Mntra)	P	P	P	P	P	P	A	A	A	D	D	D
# pins	24, 28	28	40, 44	40, 44	40, 44	28	20	20	40, 44	80	80	80
Max. Speed Avnil. (MHz)	16	16	12	12	16	12	24	24	24	16	16	16
Key features many have memory lock, power save modes. Unless noted, all are CHMOS process	I <sup>2</sup> C Serial Bus	PWM, I <sup>2</sup> C Serial Bus	Low Voltage/Power (1.8V to 6V) I <sup>2</sup> C Bus	83CL781 optimized for 12MHz @ 3.1V, I <sup>2</sup> C	256 Bytes EEPROM, 80C51 Pinout	Smart Card, 2K EEPROM, CCU	2.7-6V comp.(1), *flash EPROM	2.7-6V comp.(1), *flash EPROM	WD, 2.7-6V, 2K EEPROM, SPI port, *flash EPROM	**NVRAM for code/data	**NVRAM for code/data	**NVRAM for code/data

# A6

## Addresses, Phone Numbers, and Products

---

It would be foolish to think that this list will remain complete for six months, let alone for the (hopefully long) life of this book. However, from experience it would seem that some leads are better than none, and one good lead can open up to a host of others. The following information is what I have available in late 1996.

### MAGAZINES

<i>Circuit Cellar Ink</i> The Computer Applications Journal PO Box 698 Holmes, PA 19043-9613 (800)269-6301 (860) 875-2751 fax (860) 871-0411 <a href="http://www.circellar.com/">http://www.circellar.com/</a>	<i>EDN</i> 8773 South Ridgeline Blvd Highlands Ranch, CO 80126-2329 (303) 470-4445 fax (303) 470-4280 <a href="mailto:cahners.subs@denver.cahners.com">cahners.subs@denver.cahners.com</a>
<i>Embedded Systems Programming</i> Miller Freeman Publications 600 Harrison Street San Francisco, CA 94107 (415) 905-2200	<i>Midnight Engineering</i> 1700 Washington Ave. Rocky Ford, CO 81067 (719) 254-4558
<i>8051 Product Directory</i> MW Media 60 S. Market Suite 720 San Jose, CA 95113 (408) 286-4200	





# Index

---

~, 115  
--, 115  
!, 115  
!=, 115  
#, 61, 387  
#data, 49  
#data16, 49  
#define call, 154  
#define, 88, 136, 170  
#pragma NOREPAR:MS, 194  
#pragma, 140  
%, 110, 115, 285, 307  
%=, 101, 115  
&, 99, 115, 153  
&&, 114, 115  
&=, 101, 115  
(), 115  
\*, 115  
\*=, 101  
,, 115, 135  
., 115  
/, 15, 115, 285  
/=, 101  
/0, 144  
?., 115, 165  
@, 61, 387  
@Ri, 50  
\\, 115  
\\, 115  
\\=, 115  
^ 99, 115  
^=, 101  
\_, 199  
{}, 115  
|, 99  
||, 114  
|=, 101, 137  
~, 99, 115  
“, 143  
+, 115  
++, 115, 153  
+=, 101  
<, 115  
<<, 115  
<<=, 101, 165  
<=, 115  
-=, 101, 115  
=, 115  
==, 114, 115  
>, 115  
->, 115, 146  
>>, 115  
>>=, 101  
11.059 MHz, 31, 285,  
298  
12 MHz, 285, 298  
1488/89 302  
16-bit auto reload mode 286  
3-byte pointer. 162  
555, 346  
68HC11, 4  
74138, 15

- 7414, 350
- 74148, 324
- 74LS373, 39
- 8031, 42, 412
- 8031; expanded interrupts, 292
- 8048, 3
- 8051 core, 4
- 8051, 412
- 8051; internal architecture, 12
- 8052, 283, 355, 413
- 80C251, 4, 390
- 80C51EX, 390
- 80C520, 400, 416
- 80C552, 417
- 80C750, 154, 239, 288, 417
- 80C751, 38, 154, 239, 288, 351, 418
- 80C752, 169, 418
- 8255, 41, 181, 187
- 8259, 288
- 8-bit auto reload, 286
- 8-bit micro, 24
  
- A51, 177
- absolute code, 96
- absolute segment, 91, 174
- ACALL, 54, 69, 155
- ACC, 34, 35, 49
- access: random, 18
- access: sequential, 18
- accumulator (see ACC)
- accuracy, 106
- acknowledge, 320
- A-D converter, 45, 167, 282
  - added, 44
  - test, 264
- ADD, 57, 75
- ADDC, 57, 75, 102
- adder, 36
- addition: four byte, 112
- address
  - bus, 12
  - decoding, 14
  - latch enable (see ALE)
- addressing
  - modes, 60
  - external, 30
- Advin, 424
- AJMP, 53, 66, 222
- ALE, 26, 28, 39
- ALE: use as clock, 28
- algorithm, 36, 75
- alien, 193, 204
- Allen Systems, 422
- ALU, 13, 34
- American Automation, 420
- American Standard Code for Information Inter-
  - change (see ASCII)
- analog computer, 12
- AND, 36, 99
- ANL, 54, 56, 70, 74, 99
- ANSI C, 389
- Archimedes Software, 420
- architecture
  - 8051, 12
  - computer, 11
- arithmetic logic unit (see ALU)
- arithmetic operators, 106
- array pointers, 112
- array, 130
  - based, 142
  - byte, 130
  - constant, 135
  - initialization, 133
  - large, 132
  - message, 144
  - of array pointers, 142
  - of structures, 136
  - passing, 157
  - ROM, 133
  - sparse, 143
  - start in C, 131
  - two-dimensional, 132
  - unsized, 133
  - within structures, 137
- artificial stack, 130
- ASCII code, 224
- Ashling, 423
- assemble, 220
- assembler, 11
- assembling: DOS, 379
- assembly instructions
  - alphabetic order list, 370
  - numeric order list, 363
- assembly language, 49, 86
- assembly: switching to 8051, 387
- assignment, 114
  - operators, 101
- asynchronous transmission, 23
- AT directive, 91
- Atmel, 421
- auto reload, 286
- automatic variable, 171, 178, 235
- auxiliary carry, 72
- Avocet, 420, 421
  
- B register, 34
- background tasks, 308, 322
- bank switching, 300, 391

- bank-switched memory, 139
- Barnett, R. H., 249, 381, 393
- based array, 142
- based structure, 144
- based variable, 139
- BASIC, 5, 84, 151
  - compiled, 85
  - interpreted, 84
  - structured, 119
- Batch files (DOS), 381
- battery-backed RAM, 20
- baud rate, 302
- BCD, 77
  - packed, 78
- binary
  - numbers, 13
  - semaphore, 336
- Binary Technology, 423
- bipolar stepper, 152
- bit, 13, 86, 87
  - addressable memory, 61
  - fields, 87
- BIT, 91
- BITADDRESSABLE, 318
- BITBUS, 337, 410
- bitwise operator, 99
- block, 119
- Blue Earth Research, 423
- Boolean instructions, 72
- borrow flag, 76
- BP Microsys, 424
- branch, 117
  - three way, 68
- branching, 120
  - instructions, 66
- break, 123, 124
- breakpoint, 248, 265
- broad-range frequencies, 301
- buffering, 304
- build project, 228
- burn and try method, 240, 265
- burn EPROM, 220
- bms, 12, 24
  - expansion, 42
- byte, 13
  - array, 130
- Byte-BOS, 341, 421
- C language, 85, 177
- C/T, 283, 286
- C/T2, 287
- C51, 177
- Cactus Logic, 423
- call, 151
  - implicit, 164
- CALL, 69, 153
- capture and compare, 407
- capture mode, 288
- capture register, 299
- carry, 72, 109
- cast, 107, 110, 145, 169
- Ceibo, 422, 423
- central processing unit (*see* CPU)
- changes, identifying, 107
- changing drivers, 167
- char, 86, 87
  - array, 130
- character strings, 99
- chip, 4
- Chiptools, 420
- Circuit Cellar, 419
- circulating pump controller, 345
- CISC, 26
- CJNE, 53, 67
- clock, 296
- clock: real-time, 278
- CLR, 56, 71, 74
- CMX Company, 421
- CMX, 340
- code, 20, 140, 141, 162, 308
  - efficiency, 214
  - in-line, 155
  - option, 98, 107
  - space, 22, 89
- CODE, 91
- comments, 90
- communication, 334
- comparison, 67
- compile, 220
- compiler, 11
- compiling: DOS, 378
- complement, 71
- computer architecture, 11
- computer: analog, 12
- conditional operator, 122
- configuring a system, 156
- constant
  - floating, 99
  - long, 99
- context
  - switching, 65, 295, 330
  - saving, 290
- continue, 124
- control bus, 28
- COPS, 5
- Cottage Resources, 423
- count down, 216
- counter mode, 286

- counter, 218, 282
- counting semaphore, 318, 336
- CP/RL, 287
- CPL, 56, 71, 74, 99
- CPU, 12, 34
- CRC, 410
- CS, 41
- CSEFC, 100
- D-A converter, 322
- DA, 59, 78
- Dallas 520, 400
- Dallas Semiconductor, 421
- data, 21, 89, 111, 216
  - bus, 12
  - space, 22
- DATA, 91
- DB, 91
- DB-25, 302
- DB-9, 302
- DCX51, 337
- debugging, 240
  - information, 234
  - strategies, 262
- DEC, 58, 76
- decimal instructions, 77
- decimal subtraction, 78
- decisions, 117
- declaration, 178
- decoding, 26
  - address, 14
- definition, 178
- delay, 126, 152
  - loop, 32
- development environment, 41
- direct, 49
- directly addressable RAM, 50
- DIV, 58, 77, 111
- divide; in hardware, 77
- division, 106
- DJNZ, 53, 68, 125
- D-latch, 18
- DO UNTIL (Pascal), 124
- do while, 124, 125
- DOS, 221, 328, 377
- double, 86, 88
- downloading, 220, 240
- DPH, 34
- DPL, 34
- DPTR, 34, 50
- driver, 156
  - stepper, 152
  - changing, 167
- DS, 91, 139, 172, 178
- DS5000, 4
- ds51, 223, 241
- DS520, 158
- dscope, 242
- dynamic
  - memory allocation, 144
  - RAM, 20
- E<sup>2</sup>PROM, 19
- EA, 29, 289, 290
- edge triggered interrupt, 293
- EDI Corp., 125
- editor, 230
- EDN, 419
- EET monitor, 249
- efficiency, 5, 214, 296
  - software, 5
  - application, 5
  - software development, 5
- else, 120
- embedded assignment, 104, 121
- Embedded Systems Programming, 419
- embedded systems, 156
- Emulation Tech. Inc., 123
- emulators, 260
- EmuTec Inc., 425
- END, 91
- envelope detector, 324
- environment, 9
- EPROM, 19
- EQU, 88, 170
- equate, 170
- erasable programmable read only memory (EEPROM)
- ES, 289, 291
- ET, 2, 290
- ET0, 289
- ET1, 289
- events, 300, 337
  - fast, 298
  - infrequent, 299
- EX0, 289
- EX1, 289
- examples
  - mSec timer, 285
  - pump controller, 345
  - envelope detector, 324
  - mixed language math, 203
  - pulse generator, 322
  - read an A-D converter, 167
  - scan a keypad, 164
  - serial buffering, 304
  - solenoid cylinder, 321

- stepper driver assembly, 180
- stepper driver C, 186
- stepper driver mixed, 198
- switches to lights, 92
- time delay, 126
- traffic light (basic), 314
- traffic light (with wait), 316
- exclusive or, 36, 71, 99
- EXEN2, 287
- expansion, 40, 42
  - A-D converter, 44
  - interrupts, 291
  - parallel ports, 43
  - RAM, 43
- extensions to C, 87, 216, 389
- extern, 172, 178, 179, 187
- external
  - access pin (see EA)
  - addressing, 30
  - interrupt, 321
  - RAM, 89
  - stack, 157
- EXTRN, 172, 178, 180, 181
- factory-masked ROM, 19
- false, 87, 317
- fast events, 298
- fetching instruction, 26
- FIFO, 307
- first-in first-out buffer (see FIFO)
- flag, 72, 279, 290, 299, 333, 334
  - borrow, 76
- flash memory, 19
- flip-flop, 18, 281
- float, 86, 88
- floating-point
  - constant, 99
  - math, 300
  - variables, 209
- flow sensor, 349
- flowchart, 103, 117
- for(;;), 94, 216
- forever, 94
- formalizing, 342
- Forth, 5, 84
- four-byte addition, 112
- Franklin Software, 216, 420
- frequency
  - broad range, 301
  - high, 298
  - in-between, 301
  - low, 299
  - measurement, 355
- Fujitsu, 421
- function, 94, 151
  - global, 180
  - nested, 156
  - public, 180
  - put in a library, 210
  - recursive, 192
  - reentrant, 192
  - shared, 170
- GATE, 283
- generic pointer, 162
- GF0, 304
- GF1, 304
- global functions, 180
- glue logic, 17
- goto, 124
- H bridge, 153
- h, 177
- handler, 333
- hardware
  - divide, 37
  - interrupt, 321
  - multiply, 37
- header, 283
  - file, 94, 176, 210
- hex, 14, 99, 177
  - conversion: DOS, 389
  - file, 182, 222
  - format, 174, 223
- hexadecimal (see hex)
- high frequency, 298
- high-speed capture, 407
- HiTech Equipment, 423, 424
- Hitools inc, 424
- Hoffer Plastics Found, 249, 293
- hogging the processor, 278
- hours, 296
- I/O, 218
  - pins in testing, 266
  - ports, 12
- I<sup>2</sup>C bus, 24, 310, 410
- IAR Systems, 420, 421
- IC, 4
- ICE, 241, 260
- idata, 90, 141
- IDATA, 91
- identifying changes, 107
- IDL, 304
- IE, 34, 289, 290, 291
- IE0, 284
- IE1, 284
- if, 120

- implicit call*, 164
- in-between frequencies, 301
- INBLOCK, 218
- INC, 58, 76, 177
- in-circuit emulator (*see* ICE)
- include file, 94, 176, 283
- Incredible Tech, Inc., 425
- indentation, 122
- index, 131
- indirectly addressable RAM, 50
- infrequent events, 299
- in-line code, 155, 170
- INPAGE, 218
- input port, 73
- instruction
  - Boolean, 72
  - decimal, 77
  - decoding, 26
  - execution, 26
  - list, 363, 370
  - math, 75
  - timing, 32
- int, 88
- integer, 13, 88
- integrated circuit, 4
- Intel, 49, 420, 421, 423, 424
  - HEX, 174, 223
- internal
  - RAM, 21, 50
  - stack, 139
  - timer, 283
- interpreted BASIC, 81
- interpreter, 84
- interrupt, 288
  - edge-triggered, 293
  - enable register (*see* IE)
  - expanding with hardware, 291
  - external, 324
  - handler, 333
  - latency, 293, 294
  - level-triggered, 291, 293
  - priority, 293
  - regular, 296
  - request (*see* IRQ)
  - service routine (*see* ISR)
  - testing, 264, 266
  - vectored, 289
- interrupt4, 305
- interval, 298
- Intronics, 424
- IP, 34, 294
- IRQ, 290
- ISR, 289
- ITO, 281
- ITI, 284
- iterative loop, 125
- JB, 54, 67, 120
- JBC, 54, 67
- JC, 54, 67, 120
- JMP, 53, 67
- JNB, 54, 67, 120
- JNC, 54, 67, 120
- JNZ, 53, 67, 120
- job, 273
- JZ, 53, 67, 120
- Keil Software, 9, 216, 420, 427
- keypad, 109
  - scan, 164
- keyscan, 103
- Kontton, 421
- large, 195, 197
- latency, 276, 293, 294
- Lauterbach, 424
- LCALL, 54, 69, 154, 155
- LCD display, 32, 45, 158
  - bus driven, 46
  - port driven, 45
- learning curve, 343
- LED, 357, 359
- level-triggered interrupts, 291, 293
- LIB, 177
- librarian, 210
  - DOS, 384
- library, 174, 208
- functions, 210
  - making your own, 209
- Link Instruments, 425
- linked list, 143
- linker options, 235
- linking, 217, 220, 222
  - DOS, 380
- liquid crystal display
  - (*see* LCD)
- lists, 334
- LJMP, 53, 66, 154, 222
- LM34, 349
- LNK, 177
- load-time locatable code, 222
- locate, 220
- Logical Devices Inc., 425
- logical
  - operators, 99, 113
  - test, 114
- Logical Sys. Corp., 425
- long constant, 99

- long, 86, 88, 300
- lookup table, 107, 133
- loop, 117
  - iterative, 125
  - constructs, 124
- low frequencies, 200
- lsh, 13
- LST, 177
- M51, 177
- machine
  - code, 10
  - cycle, 32
  - instruction list, 363, 370
- macro, 170
- main, 179
- make files, 237, 228
- map file, 185
- masking 114, 290
- math
  - instructions, 75
  - floating-point, 300
  - multibyte, 73
- MATH.H, 216
- Matra, 422
- Maxim, 232, 302
- MCB520, 260, 400
  - schematic, 404
- memory, 12, 18
  - bank-switched, 139
  - battery-backed, 20
  - dynamic allocation, 144
  - EEPROM, 19
  - EPROM, 19
  - expansion, 39
  - flash, 19
  - internal, 21
  - model, 139, 233
  - off-chip, 21, 139
  - paged, 29
  - RAM, 18
  - ROM, 18
  - spaces, 138
- message, 145, 334, 337
  - arrays, 144
- MetaLink, 424
- microcomputer, 4
- microcontroller, 4
  - stand-alone, 38
- Micromint, 423
- microprocessor, 4
- Microtek, 424
- Midnight Engineering, 419
- minicomputer, 4
- minutes, 296
- mixing languages, 190
  - example, 198
  - math, 203
- mnemonics, 10, 40
- MOC3030, 358
- mod, 110
- Modular Micro Controls, 423
- modular programming, 174
- module, 174
- modules, 110
- MON51, 256
- monitor, 240, 246
  - advantages, 247
  - drawbacks, 247
  - how it works, 247
- MONITOR-51, 246
- Motorola S records, 223
- MOV, 50, 56, 61, 62, 74
- MOVC, 52, 63, 134, 179
- MOVX, 30, 51, 52, 64, 139, 332
- msb, 13, 87
- MUL, 58, 77, 111
- multibyte math, 73
- multiple indirection, 142, 143
- multiplexing, 21
  - address/data, 39
- multiplication, 106
- multiply: in hardware, 37
- MULTITASK1, 339
- multitasking, 168, 269,
  - 273, 275, 328, 352, 356
  - cooperative, 275
  - priority-based preemptive, 277
  - round robin, 275
  - time slice, 276
- names: functional, 118
- NAND gate, 14
- Needham Electronics, 261, 425
- negation, 15
- nested
  - blocks, 121
  - functions, 156
  - structures, 137
- neural net, 12
- New Micros Inc., 423
- nibble, 13
- ninth-bit mode, 310
- n-key rollover, 272
- Noahu, 424
- NOP, 54, 70
- NOREGPARMS, 194, 204
- NOT, 99



- null, 126, 143, 144
  - nybble, 13
- object file, 222
- octal, 99
- off-chip
  - access, 30
  - memory, 21, 139
  - variables, 217
- Ok! Semiconductor, 422
- on-chip stack, 331
- ones complement, 37
- operating system, 329, 330
  - commercial, 337
- optimization, 232
- options, 230
- opto-isolated triac driver, 321
- OR, 36, 99
- ORG, 96, 222, 228
- originate (*see* ORG)
- ORI, 55, 56, 71, 74, 99
- oscillator, 31
- output port, 23
- overflow, 72, 73, 109, 113, 206
- overhead, 343
- overlying, 172, 178, 217
- overload, 326
  
- P code, 220
- packed BCD, 78
- page, 29
- parallel port, 22, 41
- parameter passing, 157, 158, 191
  - alien, 193
  - fixed on-chip, 194
  - off-chip, 195
  - reentrant off-chip, 197
  - reentrant on-chip, 196
  - register-based, 193
- partial segment, 174
- passing arrays, 157
- PBIO, 261
- PC, 26, 50
- PCON, 34, 304
- PD, 304
- pdata, 30, 90, 141
- period, 135
- Phillips, 422, 425
- PL/M, 5, 84, 85, 192, 203, 388
- PLD, 17, 152
- pointer, 139, 308
  - 3-byte, 162
  - array, 142
  - generic, 162
  - universal, 141
- polling, 304
- pools, 334
- POP, 52, 65, 66
- port, 12, 22, 92
  - initial setup, 23
  - input, 23
  - masking, 114
  - output, 23
  - parallel, 22
  - serial, 23, 302
  - test, 263
- portability, 192
- power supply, 358
- precedence, 114
- preemption, 274, 278, 330
- preemptive multitasking, 277
- printf, 389
- priority, 274
  - of interrupts, 293
  - setting, 332
  - third in hardware, 293
- priority-based multitasking, 277
- PRJ, 177
- procedure, 151
- processor
  - hogging, 278
  - overload, 326
  - test, 263
- Production Languages Corp., 420
- program 174
  - counter (*see* PC)
  - development, 219, 220
  - relocatable, 174
  - status word (*see* PSW)
  - storage, 20
  - store enable (*see* PSEN)
- programmable logic device (*see* PLD)
- programmers' PROM, 261
- project, 273
  - files, 224
- PROM programmers, 261
- promote, 107
- prototype, 94, 171, 178, 179, 187
- PS, 294
- PSEN, 26, 29, 41
- pseudo code, 220
- PSW, 33, 34, 294
- PT0, 294
- PT1, 294
- PT2, 294
- PU552, 393

- schematic, 395
- public functions, 180
- PUBLIC, 172, 178, 179, 180, 181
- pulse generator, 322
- pulse-width modulation, 406
- pump drive, 358
- PUSH, 52, 65
- PWM, 406
- PX0, 294
- PX1, 294
- queue, 337
- quotient, 111
- RAM, 18
  - battery-backed, 20
  - bit addressable, 61
  - directly addressable, 59
  - dynamic, 20
  - expansion, 43
  - indirectly addressable, 59
  - internal, 21, 89
  - static, 20, 41
- RAM(large), 195
- random access memory (*see* RAM)
- random access, 18
- RCLK, 287
- read only memory (*see* ROM)
- readability, 94
- real-time, 272
  - clock, 278, 296, 314, 351
  - interrupt, 276
  - operating system (*see* RTOS)
- receiver, 304
- reentrant, 196, 204, 290, 327
  - function, 192
- reg51.h, 176
- reg51.inc, 177
- regions, 334
- register, 18
  - bank, 33, 61, 295, 330
  - special function, 21
- regular interrupt, 296
- regulator, 358
- reload, 286
- relocatable segments, 91, 96, 139, 174, 181
- remainder, 109, 110
- REN, 303
- resources, 334
- RET, 54, 69, 70, 153
- RETI, 54, 70, 289, 290, 293, 326
- return, 151, 163, 164
  - conventions, 191
  - returning values, 163
- RISC, 27
- RL, 56, 72, 100
- RLC, 56, 72, 100, 105
- Rn, 50
- rollover, 272
- ROM, 18
  - array, 133
  - ROM (small), 154, 239
- Rostek, 422
- rotate, 36, 71, 100, 216
- round-robin, 331
  - multitasking, 275
- RPM, 282
- RR, 56, 72, 100
- RRC, 56, 72, 100
- RS-232, 302, 310
- RS-422, 310
- RS-485, 310
- R5EG, 96, 181
- RTOS, 328
  - benefits, 342
  - Byte-BOS, 341
  - CMX, 340
  - commercial, 337
  - costs, 343
  - DCX51, 337
  - RTX, 339
  - summary, 341
  - USX, 339
- RTX, 339
- S records, 223
- saving context, 290
- sbit, 176
- SBUF, 34, 307
- scanf, 389
- Scanlon Design, 425
- scheduler, 276, 312, 356
- Schmidt trigger, 350
- SCON, 34, 303
- scope: of variables, 170, 171, 172, 178
- SDLC, 410
- seconds, 296
- segment, 175
  - absolute, 91, 174
  - complete, 174
  - partial, 174
  - relocatable, 91, 96, 174, 181
- SEGMENT, 181
- semaphore, 318, 334
  - binary, 336
  - counting, 318, 336

- send, 31, 256
- sequential access, 18
- serial
  - buffering, 304
  - port, 23, 302
- SETB, 56, 74
- setting priority, 322
- SFR, 21, 33, 176, 281, 283
- shared
  - functions, 179
  - variables, 178, 320, 335
  - variables: drawbacks, 335
- shift register mode, 310
- shift, 36, 101, 216
- shifting, 71
- shift-register, 36
- shortcuts, 214
- Siemens, 422
- signed, 86, 87
  - integer, 37
  - math, 37
- Signum Systems, 424
- Silicon Systems, 422
- simulator, 240, 264
  - DS51, 241
- sixty-line rule, 155, 156
- sizeof, 115, 144
- SIMP, 53, 66
- SM0, 303
- SM1, 303
- SM2, 303
- small, 139, 216
- SMOD, 302
  - doubling, 304
- software
  - development: efficient, 5
  - efficient, 5
- Solenoid cycler, 321
- SP, 34
- space-reserving files, 382
- sparse array, 143
- special function register (see SFR)
- speedometer, 282
- stack, 65
  - artificial, 139
  - external, 157
  - internal, 139
  - on-chip, 331
  - pointer, 65
- stand-alone microcontroller, 38
- Standard Microsys. Corp., 422
- static, 152, 171, 172, 178, 180, 235
  - in a function, 179
- RAM, 20, 41
- STDIO.H, 216
- stepper
  - bipolar, 152
  - driver in assembly, 180
  - driver in C, 186
  - driver, 152
  - precedence, 153
- storage program, 20
- strategies for debugging, 262
- string, 143, 144, 308
  - character, 90
- strlen, 144
- structure, 134, 322
  - based, 144
  - bit, 135
  - holding arrays, 137
  - nested, 137
  - tag, 135
  - template, 135
  - union of, 146
- structured
  - BASIC, 119
  - language, 9, 85, 118
- style, 94, 96, 118
- SUBB, 57, 76
- subroutine, 104, 151, 153
- subtraction, 36
  - decimal, 78
- suspension depth, 336
- SWAP, 59, 64, 78
- switch, 122
- synchronization, 334
- synchronous transmission, 23
- system
  - configuring, 156
  - embedded, 156
- Systonics, 420, 423
- T0, 283
- T1, 283
- T2CON, 287
- tag, 135, 138, 145, 146
- Tasking, 420
- tasks, 273, 274
  - background, 308, 322
- TB8, 303
- TCLK, 287
- TCON, 34, 284
- Tech Tools, 425
- Temic, 421
- temperature
  - conversion, 133

- sensor, 349
- template, 135, 145
- test
  - A-D, 264
  - interrupts, 264, 266
  - ports, 263
  - processor, 263
  - timers, 264
  - using I/O pins, 266
- TF0, 284
- TF1, 284
- TF2, 287
- thermistor, 349
- thermocouple, 349
- thinking, change required in, 371
- third priority, 293, 331
- three-terminal regulator, 358
- three-way branch, 68
- time delay, 32, 126
- time-slice multitasking, 26
- timer, 277, 282
  - example, 285
  - internal, 283
  - test, 264
  - watchdog, 287, 406
- timer2, 286
- time-shared systems, 276
- time-stamp method, 302
- timing, 29
  - instruction, 32
- TMOD, 34, 284, 286
- top-down programming, 179
- TR0, 284
- TR1, 284
- TR2, 287
- traffic light
  - (basic cycle), 314
  - (called function), 319
  - (with walk), 316
- transformer, 358
- transmitter, 304
- triac, 321, 358
- Tribal Microsys., 425
- tri-state, 24
- true, 87, 317
- truncating, 108
- truth table, 16
- twos complement, 36
- type, 115
  - conversion, 107
- typedef, 136
- UART, 24, 302
- uchar, 88, 133
- uint, 88
- underline, 199
- understandability, 154
- union, 145
  - of structures, 146
- unipolar stepper, 152
- UNIT, 218
- units, 318, 336
- universal asynchronous transmitter
  - receiver (see UART)
- universal pointers, 141
- unsigned, 87
  - char, 86
  - int, 86
- update project, 228
- US Software, 421
- using, 295
- USX, 339
- uVision 219, 377
  - installing, 228
- variables, 86
  - automatic, 171, 178, 235
  - based, 139
  - floating point, 209
  - off-chip, 217
  - scope, 170, 171, 172, 178
  - shared, 320, 335
- vectored interrupt, 289
- V-F converter, 282, 286
- void, 152, 164
- voltage to frequency converter
  - (see V-F)
- watchdog timer, 287, 406
- while, 124
- while(1), 94
- Widmer, NS, 249
- Windows 328, 377
- wired-OR, 324
- word, 13
- WSI Inc., 422
- XBYTE, 90, 99, 104, 161
- XCH, 52, 64
- XCHD, 59, 77
- xdata, 30, 89, 140, 141, 162, 308
- XOR, 36
- XRL, 55, 71, 99